UNIVERSIDADE D
COIMBRA

Zenaida Da Costa Adrião

RESILIÊNCIA E SEGURANÇA DE SERVIÇOS ATRAVÉS DO PARADIGMA SERVICE MESH

Dissertação no âmbito do Mestrado em Segurança Informática, orientada pelo Professor Doutor Bruno Sousa, coorientada pelo professor Doutor Nuno Antunes apresentada ao Departamento de Engenharia informática da Faculdade de Ciências e Tecnologia da Universidade de Coimbra.

Setembro de 2022

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE D
COIMBRA

DEPARTAMENTO DE ENGENHARIA INFORMÁTICA

Zenaida Da Costa Adrião

Resiliência e Segurança de Serviços através do paradigma Service Mesh

Dissertação no âmbito do Mestrado em Engenharia Informática, orientada pelo Professor Doutor Bruno Sousa, coorientada pelo professor Doutor Nuno Antunes apresentada ao Departamento de Engenharia Informática da Faculdade de Ciência e Tecnologia da Universidade de Coimbra.

Setembro 2022

Esta página foi intencionalmente deixada em branco

Agradecimentos

No percurso do desenvolvimento deste trabalho, foram muitas pessoas que deram o seu contributo para a sua consecução.
Primeiramente aos meus pais, que confiaram em mim, e deram-me essa oportunidade de estudos, a todo o apoio económico, a força e o carinho que prestaram ao longo de toda a minha vida académica, bem como, sem o apoio deles não teria sido possível.
Aos meus amigos e colegas que de uma forma direta ou indireta, contribuíram ou auxiliaram nesse desenvolvimento, pela paciência, atenção e força que prestaram em momentos menos fáceis.

Agradeço também aos meus orientadores, pela disponibilidade, e por todo o apoio.
A todos os meus professores pela paciência, pela partilha de conhecimentos, pelos ensinamentos para a vida.

Foi um grande desafio, desde sair do meu país para continuar os meus estudos fora, e pela adaptação num país completamente diferente do meu. Mas com muito esforço e com ajuda de todos que estiveram ao meu redor, tudo foi possível.
Sendo assim, a todas essas pessoas merecem, da minha parte, um sincero e profundo agradecimento.

Resumo

As tecnologias da *service mesh* tem vindo a percorrer um longo caminho desempenhando assim um papel muito importante na adoção nativa na nuvem de muitas empresas e organizações, fornecendo recursos importantes como conectividade, confiança, monitoramento e segurança. Tornando assim um elemento central da modernização de infraestruturas e Tecnologias de Informação (TI) nas organizações.
Com o avanço dos paradigmas de virtualização, e do desenvolvimento da tecnologia para microservicos a fim de melhorar significamente a velocidade e agilidade dos serviços. Têm permitido a implementação de serviços, fundamentado pela arquitetura de microserviços, que aumentou também a complexidade operacional relacionada com os aplicativos modernos. E com isso veio o surgimento do *service mesh*.
O *service mesh* é uma tecnologia que surgiu com a difusão da arquitetura de microserviços, pela facilidade de separação da rede lógica dos projetos, permitindo a concentração central na competência do aplicativo. Que se encontram distribuídos em vários servidores, containers, tornando-os dependentes da rede.

Ao longo desse trabalho vai ser introduzido conceitos e diferentes abordagens do *service mesh* que vão ser detalhadas e aprofundadas ao longo dessa dissertação.
Concretamente uma breve introdução do *service mesh*, nomeadamente os mecanismos de segurança utilizada, assim como alguns desafios. Assim como o levantamento e comparação entre as diferentes abordagens da *service mesh*, entre eles o istio, Kuma, linkerd, consult, etc.
Assim como a abordagem da arquitetura da *service mesh*, juntamente com algumas das suas vantagens e desvantagens. E por fim uma demonstração da *service mesh* no Kuma, face às diretrizes das recomendações de segurança da NIST para construção de aplicações e microserviços baseados em *service mesh* de uma forma segura.

O desenvolvimento desta dissertação, tem como objetivo principal o desenvolvimento, análise e validação de resiliência e segurança de serviços através do paradigma *service mesh* em uma *framework* do Kuma. Neste estudo foi analisado o estado da arte sobre a *service mesh*, nomeadamente a sua arquitetura, políticas de segurança, levantamento de alguns analises de vulnerabilidade, recomendações de segurança, especificamente para a framework do Kuma.

Com todo o levantamento teórico do *service mesh*, fez se também uma análise de avaliação de serviços inseguros no Kuma.

Palavras-Chave

Service mesh, microservicos, comunicação entre serviços, Kuma, segurança, resiliência, tecnologia.

Abstract

Service mesh technologies have come a long way thus playing a very important role in the cloud native adoption of many companies and organizations, providing important features like connectivity, trust, monitoring and security. Thus making it a central element of the modernization of infrastructures and Information Technologies (IT) in organizations.
With the advancement of virtualization paradigms, and the development of technology for microservices in order to significantly improve the speed and agility of services. They have enabled the implementation of services, based on the microservices architecture, which has also increased the operational complexity related to modern applications. And with that came the emergence of the service mesh.
The service mesh is a technology that emerged with the spread of microservices architecture, due to the ease of separating the logical network from the projects, allowing the central concentration on the application competence. Which are distributed in several servers, containers, making them network dependent.

Throughout this work, concepts and different approaches to service mesh will be introduced, which will be detailed and deepened throughout this dissertation.
Specifically, a brief introduction of the service mesh, namely the security mechanisms used, as well as some challenges. As well as the survey and comparison between different service mesh approaches, including istio, Kuma, linkerd, consult, etc.
As well as the service mesh architecture approach, along with some of its advantages and disadvantages. And finally, a demonstration of the service mesh in Kuma, against the guidelines of the NIST security recommendations for building applications and microservices based on service mesh in a safe way.

The development of this dissertation has as main objective the development, analysis and validation of resilience and security of services through the service mesh paradigm in a Kuma framework. In this study, the state of the art on the service mesh was analyzed, namely its architecture, security policies, survey of some vulnerability analysis, security recommendations, specifically for the Kuma framework.

With all the theoretical survey of the service mesh, an evaluation analysis of insecure services in Kuma was also carried out.

Keywords

Service mesh, microservices, inter-service communication, Kuma, security, resilience, technology.

Índice

Tabela de Abreviaturas e Siglas

API	Application Programming Interface
AWS	Amazon Web Services
ACM	AWS Certificate Manager
CA	Certification Authority
DNS	Domain Name System
EKS	Amazon Elastic Kubernetes
ECS	Amazon Elastic Container Service
EC2	Amazon Elastic Compute Cloud 2
HTTP	Hypertext Transfer Protocol
HTTP/2	Hypertext Transfer Protocol 2
IAM	Identity and Access Management
GRPC	Google Remote Procedure Call
mTLS	Mutual Authentication
SMI	Service Mesh Interface
TLS	Transport Layer Security
TCP	Transmission Control Protocol
VM	Virtual Machine

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Neste capítulo será realizada a contextualização do tema de pesquisa e a apresentação da estrutura da planificação do documento e dos objetivos propostos que contribuem para o desenvolvimento desta dissertação.

1.1 Contexto

A tecnologia da *service mesh* tem vindo a ter um grande impacto nas organizações de TI. Os desenvolvedores começaram a construir sistemas distribuídos utilizando uma abordagem multilíngue que precisa de descoberta de serviços dinâmicos, neste caso, as operações começaram a utilizar infraestruturas efémeras, de modo a lidar com inevitáveis falhas de comunicação e impor políticas de rede.
Assim, começaram a adotar sistemas de orquestração de containers como o kubernetes, monitorando dinamicamente o tráfego dentro e ao redor do sistema através de proxies e de redes modernos por API como o envoy.

Cada vez mais serviços, e aplicações são implementadas em *containers* recorrendo a plataformas como o *kubernetes* para a sua orquestração. Os *kubernetes* são um sistema de *containers open-source* que automatiza a implementação, monitorização e orquestração de aplicações em containers [1].
Com o aumento dos microserviços, criam-se desafios de como aplicar e padronizar o encaminhamento entre os vários serviços, versões, autenticação, balanceamento de carga, em clusters *kubernetes*.

Com isso, veio o conceito de *service mesh,* uma tecnologia *open-source*, composto por um conjunto de proxies configuráveis com recursos integrados para lidar com a comunicação entre os serviços e a resiliência por meio de configurações [2].

Projetada para gerenciar a comunicação entre serviços, através de uma camada de infraestrutura distribuída, criada para controlar a comunicação serviço a serviço numa arquitetura de microserviços, de forma segura e confiável, tornado mais fácil otimizar a comunicação entre os serviços, evitando assim, o tempo de inatividade à medida que os aplicativos vão crescendo.

Particularmente um aplicativo proveniente da nuvem consiste em grandes números de microserviços que podem ser implementados utilizando diferentes linguagens, assim como podem pertencer a locais distintos. Em ambientes dinâmicos depurar aplicativos de microserviços acaba por ser uma responsabilidade desafiadora [28], devido à complexidade das dependências de serviços e sem contar que qualquer serviço pode ficar temporariamente

inacessível para os seus clientes [3].

Além disso, o comportamento dos aplicativos dependem do fluxo de tráfego entre os microserviços, e com isso, o controlo de tráfego torna-se necessário para a operação em tempode execução.

Como parte do âmbito dos microserviços, a abordagem do *service mesh*, declara tratar de questões relacionadas com a comunicação, por exemplo, interoperabilidade, fluxo de tráfego, controlo de dependências [4, 5].
Diferente de outros sistemas de gerenciamento de comunicação, a *service mesh*, é uma camada de infraestrutura que dedica, diretamente numa aplicação. Isto é, cada componente da aplicação é um "serviço", ou seja, depende de outros para servir os pedidos dos utilizadores.
As *malhas de serviços* não adicionam novas funcionalidades ao ambiente de execução do aplicativo. Em qualquer arquitetura os aplicativos precisam de regras que definam como as solicitações são remetidas do ponto A e recebidas pelo ponto B. A *service mesh* difere porque requer lógica para controlar a (comunicação) intra serviço no âmbito individual e transfere para uma camada de infraestrutura.

Para que seja possível, a rede de serviço deve ser integrada ao aplicativo como uma série de proxies de rede, facilitando a otimização da comunicação entre serviços, de modo a evitar o tempo de inatividade, conforme a aplicação vai evoluindo.

Por conseguinte, houve necessidade de uma camada de infraestrutura dedicada aos microserviços que não impõe alterações nas implementações dos serviços, e que também sirva de plataforma de comunicação entre os serviços e ser gerenciável.
Para ajudar a resolver alguns desses problemas, surgiu o *service mesh*, uma abordagem capaz de mitigar essas situações, incorporado de uma camada de infraestrutura dedicada aos microserviços sem a necessidade de modificações nas implementações de serviço.

Além disso, uma *service mesh*, separa os aplicativos dos clientes internos, fornecendo localização, encaminhamento dinâmico das solicitações internas de serviço a serviço, independente de onde estejam expostas na rede. Fornece também comunicação de rede interna homogeneizadas entre os serviços, disponibilizando assim tolerância a falhas, segmento de tráfego, monitoramento, controle de dependências, aplicação de política de tempo.
No *service mesh*, as solicitações são encaminhadas entre os microserviços através da utilização dos proxies numa camada da própria infraestrutura.

Assim dentro deste cenário de evolução, o paradigma do *service mesh*, a adoção desta abordagem apresenta um certo nível de complexidade, que exige um alto custo de manutenção, assim como, requer uma mão de obra especializada, caso contrataria uma má configuração pode levar a problemas de segurança.

Pretende-se com a *service mesh*, um potencial nos microserviços, expandindo a escalabilidade, as suas funcionalidades, e a eficiência nas comunicações. Assim como, permitir que os desenvolvedores possam focar em agregar mais valores aos negócios, em vez de ocupar com a conexão entre os serviços. Facilidade em diagnosticar os problemas, como também, permitir que as aplicações fiquem mais resilientes a falhas que afetam a disponibilidade, sendo que a

service mesh tem a capacidade de verificar os componentes/microserviços em falha e encaminhar as solicitações para os componentes/microserviços totalmente funcionais.

1.2 Motivação

A motivação pelo estudo deste tema, está relacionada com a importância deste novo paradigma inovador que promete soluções para grandes organizações como, por exemplo a Netflix, denominada de *service mesh* que veio evoluindo juntamento com o avanço da tecnologia, capaz de fornecer rapidez, e segurança.

Com o avanço da tecnologia, soluções digitais estão sendo desenvolvidas em plataformas modernas, implementadas em infraestruturas de nuvem gerenciadas por plataformas de *cointainers* e com os desafios envolvidos na implementação e gerenciamento dos microserviços surgiu a criação do *service mesh.*

Embora a *service mesh* ainda esteja na sua fase de crescimento, é uma abordagem inovadora que muitas empresas estão adotando, para suprir as suas necessidades em termos de resiliência, escalabilidade e segurança dos seus serviços. O paradigma de *service mesh* é importante no desenvolvimento de microserviços dado os benefícios de abstrair a implementação e gestão demecanismos de segurança (entre outros) em aplicações de grande escala.

Neste contexto, no desenvolvimento dessa dissertação, pretende-se através do estudo da *service mesh*, mais especificamente através da framework do Kuma, sendo uma das diversas abordagensda *service mesh*. Foi escolhida o Kuma, uma plataforma de código aberto e simples, em relação aos demais que são mais complexos para o estudo e analise da segurança e resiliência de serviços.

E para esta fase de análise e avaliação da performance do serviço, o kuma corresponde às nossas necessidades.

O objetivo é, através do Kuma, analisar a performance de como os serviços comunicam e trocam informações entre eles, de uma forma segura num ambiente prático identificando as suas limitações numa perspetiva de segurança, mas também de transformação dos processos de desenvolvimento e integração de microserviços.

1.3 Objetivos

Este trabalho, centra-se no estudo da aplicação do paradigma da *service mesh* para a resiliência e segurança de microserviços. Este estudo inclui a implementação e avaliação da utilização da *service mesh* em microserviços, considerando os seguintes aspetos:

- Impacto na performance das aplicações (comparação do funcionamento do microserviços sem o *service mesh).*
- Impacto da monitorização do funcionamento dos microserviços.

- Impacto das configurações de segurança como *mTLS.*
- Impacto na performance de microserviços com diferentes políticas de tráfego.
- Impacto na resiliência e performance de microserviços na presença de falhas e (possíveis ataques).

O desenvolvimento deste trabalho para alcançar o objetivo final propõe-se na seguinte forma:

1- **Desenhar uma *framework*** baseada em *service mesh* para a resiliência e segurança de microserviços, que compreende em:
- Identificar as diferentes abordagens da *service mesh* (Istio, Linkerd, Kuma...);
- Identificar os mecanismos de modelação de serviços para múltipla arquitetura;
- Experimentar implementações de *service mesh* com o Kuma*;*

2- **Implementar funcionalidades** de serviços com complexidade variável e com elevados requisitos de fiabilidade e segurança em Kuma.

3- **Avaliar a performance** de serviços com a *service mesh*.

4- **Documentação** dos resultados para efeitos de dissertação.

1.4 Estrutura e Organização do Documento

Este documento está estruturado da seguinte forma:

Capítulo 2: apresentação do estado da arte, com realce nos conceitos do paradigma *service mesh*, a sua arquitetura, assim como todos os outros mecanismos que o constitui.

Capítulo 3: aborda a análise dos protocolos de segurança em *service mesh,* assim como um conjunto de recomendações e boas práticas implementadas no uso da *service mesh.*

Capítulo 4: aborda o estudo da análise detalhada do Kuma que inclui o seu funcionamento, suas dependências, políticas e gestão dos certificados de modo a avaliar a performance dos serviços com a *service mesh* através da framework Kuma.

Capítulo 5: inclui a análise do estado da arte relativamente à modelação dos serviços do Redis e da aplicação de demonstração do Kuma, em uma experimentação no Kuma.

Capítulo 6: inclui a apresentação do componente da parte prática, que contém a implementação e configuração de um conjunto de serviços com complexidade variável no framework do Kuma.

Capítulo 7: neste capítulo é apresentado a avaliação da performance dos serviços.

Capítulo 8: neste capítulo são apresentadas as conclusões e os desafios encontrados ao longo do desenvolvimento desta dissertação.

Capítulo 2

Service Mesh

Neste capítulo é apresentado o estado da arte do paradigma *Service Mesh*, onde será abordado a contextualização do tema *Service Mesh*, fazendo levantamento dos conceitos, da sua arquitetura, das diferentes abordagens do *Service Mesh*, e dos mecanismos.

2.1 Introdução e Conceitos

De acordo com *The new stack [33]*, a *Service Mesh* é uma tecnologia que tem por objetivo "melhorar a segurança, monitoramento e controle de tráfego de sistemas distribuídos", ou seja, um conjunto de ferramentas adaptadas a *containers*, em arquiteturas de microserviços e geridos por plataformas como o kubernetes.

O conceito de *Service Mesh* fornece serviços de infraestrutura para todos os aplicativos a partir da arquitetura de microserviços, através de configurações e de baixa latência delineada para operar com um grande volume de comunicações entre os diferentes serviços, e instância para cada um desses serviços.

Os microserviços são uma grande vantagem, pois compõem uma arquitetura que divide os seus aplicativos em vários serviços mais pequenos, que implementam funcionalidades mais simples, mas que posteriormente integradas possibilitam o aplicativo como um todo.

A essência dos microserviços é a comunicação serviço a serviço. Não há necessidade de uma camada de malha de serviço para que a comunicação possa ser codificada em cada serviço, mas à medida que a complexidade da comunicação aumenta, as malhas de serviço estão se tornando cada vez mais integradas. Para aplicativos nativos de nuvem construídos numa arquitetura de microserviços, uma malha de serviço é uma forma de incluir um grande número de serviços diferentes num aplicativo de trabalho.

Service Mesh, é o termo geralmente utilizado para descrever uma rede de microserviços que compõem todos os aplicativos e as interações entre eles, evitando que os desenvolvedores dos aplicativos/serviços, tenham que implementar segurança ou disponibilidade dos serviços separadamente a cada aplicativo, separando a complexidade incorporado pelas bibliotecas ou componentes de terceiros, através de uma camada separada denominado de proxies de serviço, que fornecem utilidades, como gerenciamento de tráfego, autenticação, monitoramento, e segurança [24].

Neste contexto, a *Service Mesh* apresenta um conjunto de três objetivos fundamentais na comunicação entre os serviços nomeadamente a ligação, a segurança e monitoramento de microserviços [34]:

- **Ligação**: *Service Mesh* permite que os microserviços comuniquem entre si, permitindo descobertas de serviços através dos proxies que são utilizados em uma *service mesh*, que reconhece a camada do aplicativo, que normalmente opera na camada 7 na pilha da rede OSI. Isso significa um encaminhamento inteligente e dinâmico assim como a resiliência de comunicação através de fluxos do tráfego e o rotulo de métrica que são baseadas nos dados do cabeçalho *http*.

- **Segurança**: a *service Mesh* também permite comunicação segura entre os seus serviços, impondo políticas de segurança, que permitem ou negam a comunicação.

- **Monitoramento**: a *Service Mesh* permite fazer a monitorização do sistema de microserviços distribuído.

Estes recursos proporcionam controlo operacional que fornece a capacidade de observação do comportamento de toda a rede de microserviços distribuídos.

2.2 Mecanismos de segurança utilizadas em service mesh

Ambos os protocolos Transport Layer Security (TLS) e Mutual Transport Layer Security (mTLS), são protocolos de segurança que fornecem comunicações encriptadas e autenticadas na Internet. Estas definições de segurança definem a autenticação como o ato de estabelecer algo como autêntico, ou seja, de como que o dispositivo se identifica na rede e a encriptação é como que os dados são encriptados ao serem enviados para a rede, isto é, um processo de transformar a informação utilizando um algoritmo de modo a impossibilitar que os não autorizados tenham acesso à informação. Esses protocolos utilizados pela *service mesh* para proteger a comunicação entre os microserviços ajudando assim a evitar ataques de espionagem, adulteração e falsificação de mensagens e ataques *man-in-the-midle*, um ataque do qual o invasor redireciona as comunicações entre duas partes da rede por meio do computador, sem conhecimento de nenhumas das partes [40].

TLS assim como mTLS são protocolos de segurança utilizados para proteger a web, o TLS protege o tráfego por meio da identificação unidirecional através da criptografia assimétrica, onde envolve duas partes que protegem a informação através de chaves públicas e chaves primárias, permitindo assim, encriptar com uma chave pública e desencriptar com a chave privada [39].

O TLS mútuo ou mTLS basicamente é uma extensão do TLS, que estabelece conexão TLS criptografada em que ambas as partes utilizam certificados digitais X.509 para efetuar a autenticação entre as partes envolvidas [41]. Ou seja, identificação bidirecional, em que estabelece uma conexão segura entre o cliente e o servidor depois que ambos validem a autenticidade um do outro.

A *service mesh* está desenvolvendo com umas das principais arquiteturas de implementação e gerenciamento de ambientes de microserviços, oferecendo uma melhor segurança, e para proteger essa comunicação o *service mesh* utiliza estes protocolos de segurança.

2.3 Diferentes abordagens do Service Mesh

A seguir segue as diferentes alternativas para a *Service Mesh*, soluções de código aberto, cada um com os seus próprios benefícios e desvantagens são: [1]

- **Istio,**
- **AWS App Mesh,**
- **Cônsul Hashicorp,**
- **Traefik Mesh,**
- **Linkerd,**
- **Kuma**

Estas são detalhadas nas seções seguintes.

2.3.1 Istio

O **Istio**, é uma malha de serviço proveniente do kubernetes desenvolvida primeiramente pela Lyft e foi a primeira solução nativa do *kubernetes* [44].
O Istio é uma plataforma de código aberto disponível para fornecer de uma forma uniforme, a integração de microserviços, gerenciando os fluxos de tráfego entre os microserviços. O Istio apresenta um plano de controle que fornece uma camada de abstração sobre a plataforma de gerenciamento de clusters subjacente, como kubernetes.
No Istio nada mais é que um plano de controle que oferece um conjunto de recurso que nos ajuda a gerenciar toda a nossa arquitetura de microserviços, de modo que dê para fazer a gestão do tráfego, aplicar políticas de comunicação, realização de monitoramento, interrupção de comunicação e injeção de falhas. [17]

Alguns benefícios do Istio inclui [44]:

- Balanceamento de carga, com reconhecimento de latência, o que vai permitir com inteligência de acordo com o tempo de resposta de cada *backend*.

- Autenticação TLS entre todos os microserviços, isso significa que toda a requisição que entra e sai do container tem que ser encriptada, pelo proxy que posteriormente, vai desencriptar a informação e enviar para o container.

  Controle de comportamento do tráfego com regras de encaminhamento avançadas, assim como uma camada de política que conecta a uma API de configuração com suporta controles de acessos, limite de rotas e taxas.

2.3.2 AWS App Mesh

A AWS App Mesh é uma *service mesh* que disponibiliza redes para aplicativos com finalidade de promover comunicações entre serviços através de diferentes tipos de infraestrutura de computação, facilitando assim, o monitoramento, controlo e comunicação entre os serviços.
A app mesh, normaliza o modo de comunicação entre os seus serviços, disponibilizando assim, visibilidade ponta a ponta, de modo a garantir aos seus serviços uma alta flexibilidade [17].
A app mesh é incorporado com os serviços da AWS, que é uma plataforma de serviços de computação em nuvem, para monitoramento, e rastreamento que funciona com bastantes ferramentas. A app mesh pode ser utilizado com containers de microserviços que são administrados por *amazon ECS, amazon EKS, AWS Fargate,* com *kubernetes* executados em *AWS* [18].

A app mesh simplifica a consecução da visibilidade, segurança e domínio sobre os seus serviços, sem a necessidade de desenvolver códigos novos ou até mesmo de implementar uma infraestrutura adicional da AWS [18].

Com a app mesh é possível:

- Uniformizar a comunicação entre os serviços;
- Efetuar regras de comunicação entre os serviços;
- Captura de métricas, *logs* e rastreamento de modo direto em produtos AWS.

2.3.3 Cônsul hashicorp

O Cônsul hashicorp é um *service mesh* que fornece uma camada de rede distribuída em meio de proteger, conectar e configurar serviços em tempo de execução nas suas plataformas. O cônsul foi um serviço descoberto que oferece suporte aos serviços numa infraestrutura moderna de microserviços.
O *Cônsul hashicorp*, permite automatizar totalmente o trabalho da rede num ambiente *data center* de modo a fornecer possibilidades de autosserviços no aspeto de agilizar a entrega dos aplicativos, de modo a reduzir a sobre carga sobre os serviços [17].

O *Cônsul* funciona no âmbito, de um plano de controle central para qualificar uma rede em várias nuvens para ambientes dinâmicos. Através de um registo de serviços que fornecem um diretório em tempo real dos serviços que estão a ser executados [17].

2.3.4 Linkerd

O Linkerd é um dos *service mesh* de código aberto mais utilizado pelas organizações [44], como também o mais antigo do mercado, bastante semelhante ao Istio, embora mais flexível, acentuando na simplicidade e é exclusivo do kubernetes, a fim de melhorar o desempenho dos seus aplicativos *kubernetes*, oferecendo segurança, confiabilidade e capacidade de observação. O Linkerd por protótipo oferece algumas funcionalidades como [13]:

- Identificação de protocolos;
- Proxy HTTP1.1/2 e GRPC;
- Injeção automática e zero configurações;
- mTLS automático por padrão;
  O Linkerd por padrão utiliza a encriptação para todas as comunicações internas com o mTLS [14], ou seja, autenticação bidirecional, onde as duas partes envolvidas se autenticam ao mesmo tempo, num protocolo de autenticação;
- Divisão de tráfego.

2.3.5 Traefik Mesh

O Traefik mesh é uma ferramenta da *service mesh*, considerada um serviço leve e simples e de fácil utilização, moderno com um balanceador de carga que facilita implementações de microserviços.
O *trafefik mesh* suporta o SMI [35] *service mesh*, conforme o site do SMI, o objetivo é fornecer um conjunto portátil de API's de malhas de serviços para que o utilizador do kubernet possa utilizar, sem a necessidade de vincular qualquer implementação específica.
O que facilita a agregação com soluções pré-existentes.

O SMI (service mesh interface), é um padrão, do qual sem ele, cada tecnologia do *service mesh* teria de utilizar APIs diferentes, o que por consequência tornaria as coisas um pouco mais desafiadoras e demoradas para os desenvolvedores. Isso porque sem as API, os desenvolvedores teriam que elaborar os seus próprios códigos para suprir as funcionalidades aos serviços, o que exigiria mais esforço [16].

Alguns APIS SMI que ajudam no gerenciamento do tráfego *service mesh* em um cluster [16]:

- Partição de tráfego;
- Métricas de tráfego
- Controle de acesso de tráfego
- classificação de tráfego

2.3.6 Kuma

O Kuma é uma abordagem de código aberto da *service mesh* e microserviços que podem ser executados e operados em ambientes kubernetes, máquinas virtuais na nuvem ou local [4].

O Kuma oferece suporte *envoy* como uma tecnologia proxy de *dataplane* (plano de dados). Embora que não requer uma experiência em *envoy*, isto é, o Kuma é capaz de abstrair as políticas que iremos utilizar, e caso exista uma política que o Kuma não tenha suporte nativamente, pode-se utilizar a política modelo do proxy com as configurações do *envoy*.
De modo que, se houver algo que o *envoy* pode fazer, e o Kuma não, ainda assim teremos acesso ao ecossistema do *envoy* através da configuração do modelo proxy [20].

Alguns benefícios do Kuma inclui:

- Plano de controle centralizado, permite que vários *service mesh* independentes sejam operadas e controladas num lugar, reduzindo assim custos operacionais. Ao contráriodo Istio, o Kuma alega simplicidade.

- O Kuma, é uma *service mesh*, universal e nativo dos *kubernetes*, independente que pode ser utilizada e operada em qualquer lugar, apresenta uma abordagem autónoma com várias zonas, que suporta várias nuvens, clusters, kubernetes com descoberta de serviços DNS nativos [4].

- Faz o registro do tráfego da rede que chega no servidor, sustentando a decisão da escalabilidade do sistema.

A arquitetura do Kuma, é constituído por dois componentes,

- Kuma-cp;
- Kuma-dp;

O Kuma-cp, é do plano de controle, é o plano responsável por realizar todo o gerenciamento.
O Kuma-dp, é o plano de dados, é o plano responsável pela implementação da comunicação feita em cada serviço.

O Kuma está sujeito a implementações autónomas, assim como implementações em várias zonas. Para a implementação autónoma, implica que o Kuma e os seus proxies do plano de dados sejam implementados em modo topologia da rede autónomo afim de a conetividade entre os serviços de cada proxy do plano de dados seja estabelecida de forma direta para todos os outros proxies de plano de dados [4].

O modo autónomo do Kuma, dá suporte a implementações complexas de várias zonas, ou zonas híbridas (Kubernetes + VMs).
Uma das principais vantagens de um plano de controle centralizado no Kuma, é a possibilidade de ter muitas redes isoladas sendo controladas por um único elemento. O que dá suporte e

permite reduzir os custos operacionais. Ou seja, o modo autónomo do Kuma possibilita essa vantagem.

A Implementação do Kuma em multizonas é a abordagem mais avançado do Kuma, em que permite o suporte de execução em muitas zonas, abrangendo implementações híbridas em *kubernetes* e também em máquinas virtuais (VM).
A nível dessas duas implementações, que, por outro lado ambiente virtualizado podemos ter um sistema operacional inteiro e um ou mais aplicativos. Enquanto que por outro lado temos o kubernetes, uma plataforma para gerenciamento de containers. Ao nível de container, eles consomem menos recursos, assim são mais leves, isso porque num servidor de muitos aplicativos em um container, teremos apenas um sistema operacional. Com isso, em termos de ciclo de vida entrea virtualização da VM e a conteinerização do kubernetes, os containers são a melhor escolha devido ao rápido tempo de configuração e leveza, enquanto que as VM apresentam um ciclo devida mais longo, mais demorado para configurar e assim mais pesado.

Sobre a implementação de várias zonas do Kuma, permite o suporte da execução dos seus serviços em várias zonas. O seu ambiente de malhas inclui diversos serviços isolados, ou seja, multi-locação, onde cargas de trabalho a ser executadas em diferentes regiões, em diferentes *datacenters*, ou em diferentes nuvens.
Uma zona pode ser um cluster *kubernetes*, uma VM, ou qualquer implementação que inclui o mesmo ambiente do serviço distribuído.

A implementação de multizonas pelo Kuma inclui [4]:

- Plano de controlo global;
  O plano de controle permite aceitação de conexões somente do plano de controle de zona, e não de qualquer proxie de planos de dados, fornecendo configurações de políticas globais que serão executadas aos proxies de plano de dados.

- Plano de controle de zona;
  O plano de controle de zona, permite conexões de proxies de dados inicializados na mesma zona, de modo que, o plano de controle global conecte-se com o plano de dados através de políticas de proxies.

- Proxies do plano de dados;
  Os proxies do plano de dados, permite a conexão ao plano de controle de zona na mesma zona.

A implementação de multizonas do Kuma permite o gerenciamento da conetividade dos serviços, mantendo conexões entre as zonas com a entrada da zona, e com o resolvedor de DNS. Ou seja, o resolvedor de DNS, é agregado a cada proxy de plano de dados de modo, que ele resolva cada endereço de serviço para toda a comunicação de serviço a serviço.

Enquanto que o plano global e o plano de controle de zona, se comunica afim de sincronizar os recursos, como, configurações de política.

Mais detalhes sobre o Kuma serão abordados no capítulo 4.

2.4 Service Mesh e alguns dos seus desafios

O desenvolvimento de tecnologias para os microserviços afim de melhorar a velocidade e a agilidade da entrega dos serviços, aumentando também a complexidade operacional relacionada a aplicativos modernos.
A ideia central dos microserviços, é dividir um monolítico complexo em pequenos serviços com finalidade de serem implementados e mantidos de uma forma independente, projetados pelas vantagens dos microserviços como, flexibilidade, simplicidade e escalabilidade.

A *service mesh* é uma abordagem que dedica sobre os microserviços sem impor modificações nas implementações de serviços agindo como uma plataforma de comunicação de serviço a serviço de forma gerênciavel, projetada para padronizar operações de tempo de execuções de aplicativos.

Um aplicativo nativo da nuvem pode representar por um grande número de microserviços que podem ser implementados por diferentes linguagens, assim como também pertencer a diferentes locais, e ter milhares de instâncias de serviços. Em ambientes dinâmicos, depurar aplicativos de microserviços acaba por ser um encargo desafiadora [28], por conta da complexidade das dependências de serviços e também, pelo fato de qualquer serviço pode tornar temporariamente inacessível para os seus clientes.
O comportamento do aplicativo depende do fluxo do tráfego entre os microserviços, portanto o controle de tráfego se torna obrigatório, para a operação em tempo de execução.

Alguns desafios identificados com o service mesh

A visão de uma malha de serviço apresenta três desafios principais identificadas como [8]:

- **O design que suporta alto desempenho;**
  No design da *service mesh*, no nível de plano de dados, integrante do proxy co-localizado com instâncias de serviços, é o ‘’coração’’ da *service mesh*, ou seja, é o responsável por intercetar e mediar o tráfego entre os microserviços para alto desempenho.

- **Adaptação**,
  Para dar suporte e acomodar uma vasta variedade de plataformas de orquestrações nativas de aplicativos em nuvem, uma malha de serviço precisa oferecer suporte a um certo nível de com figurabilidade, escalabilidade e capacidade de conexão.

- **Alta disponibilidade,**
  A alta disponibilidade é uma preocupação, porque uma estrutura de serviço altamente disponível pode reduzir o risco da tomada de decisões devido à indisponibilidade.

A *service mesh* oferece mecanismos globais uniforme para controlar e medir todo o tráfego de solicitações entre os microserviços. Em nível de plano de dados, isto é executado atribuindo umproxy leve para cada instância de serviço [28]. Apesar de que as plataformas modernas de orquestração de containers como, por exemplo: o kubernetes e o Mesos [56], são projetadas paralidar com falhas de serviço, a disponibilidade do serviço ainda pode ser dificultada pela falha introduzida pelo proxy, a mesma situação pode ser empregue aos componentes no plano de controle, por exemplo, se o componente que é responsável pela autenticação e autorização estiver momentaneamente indisponível, o comportamento do aplicativo pode proceder a um estado inconsistente de modo que os serviços recebam solicitações que deveriam ser sinalizado como inválido.

Consequentemente, uma *service mesh*, altamente disponível é fundamental não apenas para garantir a funcionalidade prometida, mas também para garantir que as disponibilidades dos serviços de proxy não sejam corrompidas.

2.5 Comparação entre os diferentes Service Mesh

Para as diferentes abordagens da *service mesh* estudadas, eles apresentam alguns recursos em comum, como [22]:

- Suportes de protocolos, todos apresentam suporte com HTTP, HTTP/2, GRPC, TCP e Websockets.
- Todos dispõem da segurança básica mTLS entre proxies.
- Todos fornecem alguma forma de balanceamento de carga.
- Qualquer Service Mesh apresenta o recurso de descoberta de serviço.

Para as diferentes *frameworks* do *service mesh*, destacam-se algumas diferenças em nível de tráfego, monitoramento, implementação entre outros aspetos, será apresentada as seguintes comparações [22] [23] em aspetos de infraestruturas.

- *Comparação da Infraestrutura*

Tabela 1 - Comparação da infraestrutura entre os diferentes service mesh

	Istio	**Kuma**	**Linkerd**	**Cônsul**	**AWS app mesh**	**Traefik mesh**
Plataforms	Kubernetes VM(Universal)	Kubernetes VM(Universal)	Kubernetes	Kubernetes VM(Universal)	AWS EKS, ECS, Fargate, EC2	Kubernetes
Hight availability	sim	sim	sim	sim	sim	sim

for control plane						
Sidecar Proxy	sim	sim	Sim, linkerd-proxy	sim	sim	Não
Mesh expansion	sim	sim	não	sim	Sim, para serviços AWS	Não
Multi-cluster	sim	Sim (multi-zona)	sim	sim	N/A	Não

Na tabela 1, demonstra a comparação entre as diferentes *service mesh* em nível de infraestrutura.

Em nível de plataforma, todos, exceto o AWS app mesh, utilizam o kubernetes ou o VM universal, enquanto que o AWS app mesh utiliza EKC, ECS, Fargate EC2.
EKS-Elastic kubernetes, é um serviço que permite executar o kubernetes na AWS sem necessidade de instalar, operar ou manter o próprio plano de controle do kubernetes.
ECS-é um serviço de administração de containers de alto escalável e de alto desempenho, que permite a execução de aplicativos em clusters que são administradas pelo Amazon EC2.
EC2- permite a implantação de aplicações escaláveis a uma web service de modo que o utilizadorpossa trabalhar com a sua aplicação virtual.
Fargate- é uma tecnologia utilizada para executar *cointainers* sem a necessidade de fazer o gerenciamento de servidores ou de clusters [26].
O traefik mesh é o único na nossa comparação de service mesh que não suporta a injeção do sidecar proxy, em vez disso, ele é implementado como um *daemonSet* em todos os nós para operar como um proxy entre os serviços.

- *Comparação da gestão do tráfego*

Tabela 2 - Comparação da gestão do tráfego entre os diferentes service Mesh

	Istio	**Kuma**	**Linkerd**	**Cônsul**	**AWS app mesh**	**Traefik mesh**
Fault injection	Sim	Sim	Sim	Não	Não	Não
Limite de taxa (rate limiting)	Sim	Não	Não	Sim (através do envoy)	Não	Sim
Circuit Breaking	Sim	Sim	Não	Nao	Não	Não

Traffic split	Sim (através de suportes de terceiros)	Não	Sim	Não	Não	Sim
Traffic Metrics	Sim (através de suportes de terceiros)	Não	Sim	Não	Não	Não
Traffic access control	Sim (através de suportes de terceiros)	Não	Não	Sim	Não	Sim
Per-Route Metrics	experimental	Não	sim	Depende do proxy em execução	N/A	Não

O Cônsul e o Traefik mesh fornecem o controle de acesso ao tráfego através do SMI (*service mesh* interface), conjunto de recursos que permite a especificação de recursos para uma *service mesh.*

Quebra de circuito em Kuma, é uma política com finalidade de procurar erros no tráfego que está sendo trocado entre os proxies de plano de dados, com objetivo de certificar que nenhum proxy adicional atinja o plano de dados até que seja confiável.

- Comparação da observabilidade, recursos de monitoramento

Tabela 3 - Comparação da observabilidade entre os diferentes service mesh

	Istio	**Kuma**	**Linkerd**	**Cônsul**	**AWS app mesh**	**Traefik mesh**
Distributed Tracing	Sim (openTelemetry)	Sim	sim	sim	sim	sim
Monitoring with prometheus	sim	sim	sim	Não	Não	sim

Em comparação aos recursos de monitoramento, da tabela 3, todos os diferentes *service mesh* suportam rastreamento distribuído. O Istio suporta rastreamento distribuído através do open Telemetry, que é uma ferramenta utilizado para gerar, coletar e exportar dados (métricas, logs, rastreamento) ponta a ponta em arquiteturas de microserviços.

Ao monitoramento com o *Prometheus*, [36], é uma plataforma de monitoramento de *open source*, que faz a coleção e armazenamento de métricas temporais. Ou seja, uma ferramenta de monitoramento de serviços e aplicações, apenas o Istio, Kuma e Linkerd e Traefik mesh suportam.

- Recursos de segurança entre os diferentes *service mesh*

Tabela 4 - Comparação dos recursos de segurança entre os diferentes service Mesh

	Istio	**Kuma**	**Linkerd**	**Cônsul**	**AWS app mesh**	**Traefik mesh**
mTLS	Sim	Sim (não está ativo por omissão)	Sim	Sim (não está ativo por omissão	Sim	Não
Regras de autorização do serviço	Sim	Sim	Sim	Sim	Não (mas suporte para IAM)	Não
Certificados CA	Sim, cert CA pluggable and CA integration (experimental)	N/A	sim	Sim,HashiCorp, valut, ACM private CA, Custom CA	sim	não

Na tabela 4, em nível de segurança todos, exceto o traefik mesh suportam o recurso mTLS, o protocolo de segurança que permite que o tráfego da rede seja automaticamente encriptado com a identificação bidirecional para que todos os proxies tenham a sua identidade no plano de dados.

Acerca da regra de autorização do serviço, o AWS app mesh, não aplica regras de autorização de serviços, mas utiliza o suporte de IAM (*Identity and Access Management*) que faz a autorização do utilizador.
IAM [42], é um serviço que dá suporte ao AWS app mesh em como controlar com segurança quem está autorizado e autenticado para utilizar recursos como, por exemplo: credenciais temporárias.

Após uma análise comparativa entre os diferentes *services mesh*, pode-se dizer, que o melhor *service mesh* a ser utilizado, depende muito da necessidade e dos requisitos que os desenvolvedores pretendem, entendendo assim as suas habilidades, recursos e tempo que dispõe para a implementação.

A análise comparativa nos dá uma ampla de informação, que nos permite um bom ponto de partida para exploração.

Em análise do *Linkerd*, ele fez grandes progressos desde a sua primeira versão em termos de facilidade de uso assim como a facilidade de instalação, porém, ele remove a extensibilidade inerente que o *envoy* oferece. O que reflete o não suporte de interrupção de circuito, injeção de atraso, limitação de taxa.

Entre os diferentes *service mesh*, o Istio, apresenta uma maior comunidade online até o momento [22]. Oferecendo suporte em ambientes kubernetes e máquinas virtuais (VM).
Em oposição, o Istio não é gratuito, por duas vertentes: a primeira é que os seus requisitos são altos, em termos de tempo de configuração, e do tempo de o manter a funcionar corretamente, de acordo com o tamanho da infraestrutura e do número de serviços. E para oferecer um trabalho totalmente integro e funcional em tempo integral exige um tempo de execução.
A outra vertente é a adição de uma certa quantidade de sobrecarga dos recursos.
O plano de controlo do Istio só é compatível com *containers* do kubernetes, ou seja, não modos em máquinas virtuais (VM) ao contrário do plano de dados.

O Kuma, foi criado em resposta às primeiras *service mesh* desenvolvidas, por serem bastantes pesadas e complexas em exercer.
Na área de gerenciamento de tráfego, o Kuma oferece recursos como injeção de falhas e interrupção de circuito, como também a criptografia mTLS entre a comunicação entre os serviços.
Em nível de descoberta de serviço, o Kuma tem disponível com o seu próprio resolvedor de DNSem execução na porta 5653 do plano de controle.
Através do multizona, o Kuma apresenta um forte enfâse na funcionalidade do *multimesh*, ou seja, suporte de combinar diversos clusters kubernetes ou modos de VM em um cluster Kuma comum.

O Traefik mesh, em nível de recursos de gerenciamento tráfego inclui interrupção de circuitocomo também limite de taxa.
Em termos de monitoramento, o Traefik apresenta suporte de *OpenTracing*, nativo, e métricas prontas para utilização.
OpenTracing, é um padrão de monitoramento distribuído através de uma API independente que fornece facilmente o rastreamento.

AWS App mesh, possui a integração nativa com Amazon web services (AWS), ao contrário do Traefik mesh, o AWS possui suporte de *multicluster*, também como a maioria dos *service mesh*, ele é apoiado pela injeção do *envoy*, e o proxy sidecar.

2.6 Arquitetura Service Mesh

A *service mesh* num nível básico, consiste em serviços e proxies de execução como registros secundários aos serviços, que incluí uma autoridade que configura os proxies para combinar com os proxies e os serviços num sistema distribuído apropriado que inclui um plano de controle e um plano de dados. Onde todas as solicitações enviadas ou recebidas de um serviço passam por dois proxies na malha: o proxy de serviço de chamada e o proxy de serviço de recebimento.

Essa arquitetura suprime todas as funções que não estão relacionadas à lógica de negócios de serviços e aos desenvolvedores de serviços. O plano de dados faz o gerenciamento dos proxies e os serviços, e o plano de controle faz o domínio que fornece políticas e configurações para o plano de dados.

O plano de controle é responsável por:

- Registro de serviços: O plano de controle precisa ter uma lista de serviços e endpoints disponíveis para fornecer aos proxies, que compila o registo consultado o sistema de programação subjacente, como o kubernetes, afim de conseguir a lista de todos os serviços que estejam disponíveis.

Configuração de proxy sidecar: as configurações dos proxies secundários inclui políticas e configurações de toda a malha, para que os proxie tenham conhecimento de executar funções adequadamente.

2.7 Algumas considerações sobre a arquitetura do *Service Mesh*

Uma *service mesh* apresenta uma solução para muitos aspetos de conceção do sistema de *microserviços* e da implementação do sistema, porém existem algumas observações, sendo descritas a seguir, conforme o documento [9]:

- **Sobrecarga no processamento;**
  As solicitações de um *microserviço* para outro são entregues por meio de um proxy e possivelmente por um balanceado de carga. Além disso, as solicitações são rastreadase podem ser modificadas por criptografia. Embora a criptografia não cause sobrecarga significativa em cada nível, ela aumenta latência e os requisitos de recursos. Para determinar se a sobrecarga de um determinado caso de uso é relevante, analise-a em termos de desempenho e escalas de medição.

- **Complexidade de design e de configurações;**
  A criação de uma configuração de malha de serviço é uma tarefa de design que requer a garantia de que os requisitos sejam atendidos corretamente. Requer conhecimento dos recursos de configuração do *service mesh*, em geral. Assim como saber criar as

configurações corretas para aplicativos específicos. Quando as malhas de serviço são ajustadas, essa configuração precisa refletir os requisitos do sistema.

- **Validade de configurações de testes;**
  Depois que as configurações do *service mesh* estiver em vigor, ferramentas como o istioctl analyzer pode ser utilizado para validar as configurações.

- **Atualizações do plano de controle;**
  A *service mesh* é um serviço que é utilizada como um sistema que pode mudar ao longo do tempo. Por exemplo, mudanças para lidar com melhorias de desempenho, escalabilidade, recursos adicionais ou correções de bugs podem ser utilizadas. Ao atualizar o plano de controle da *service mesh*, é importante verificar a configuração da corrente reversa em relação ao sistema atualizado. O teste de reversão precisa confirmar que a nova versão do *service mesh* não afeta negativamente o comportamento da *service mesh.*

- **Verificação da configuração da *service mesh***
  A existência de um plano de controle não garante automaticamente a segurança e a confiabilidade do sistema. Uma malha do serviço precisa ser ajustada e essa configuração precisa ser testada e verificada adequadamente. Afim de evitar problemas como aplicativos inseguros que podem ser detetados.

Alterações em microserviços, como adições ou atualizações, podem alterar o comportamento da comunicação de malha, mas não na medida em que a configuração seja considerada uma alteração. Para garantir que todas as alterações sejam cobertas adequadamente pela configuração do *service mesh*, é preciso executar uma atualização de configuração para cada alteração.
A malha de serviço não cobre todos os recursos de segurança que precisam ser usados em um ambiente corporativo. A malha de serviço cobre todos os aspetos da conexão de serviço; quaisquer requisitos de segurança de infraestrutura, como firewall e proteção de rede, precisam ser abordados separadamente.

Apesar das vantagens da *service mesh*, a arquitetura da *service mesh* apresenta alguns desafios com requisitos de segurança de acordo com a documentação da NIST.

- Quanto mais microserviços, mais interconexões existem entre eles, portanto mais comunicação para manter segura;
- *A service mesh* suporta serviços em ambientes dinâmicos, o que implica implementação de descoberta de serviços seguros;
- Inexistência de perímetro de rede;
- A natureza dos microserviços requer autorizações granulares em cada microserviços, isso implica definir muito bem as políticas de segurança e configurações sejam bem definidas para cada microserviços afim de permitir a uniformidade.

2.8 Service Mesh em uma arquitetura de microserviços

Com o passar do tempo, em que a arquitetura dos serviços, sucederam de serviços monolíticos para microserviços, que consequentemente com essa mudança, o maior desafio que as organizações tiveram que enfrentar foi a divergência dos serviços e o balanceamento de carga. É importante entender do quão útil a *service mesh* pode ser. Historicamente, a infraestrutura das tecnologias de informação (TI) tradicional, mantinham as suas aplicações em arquiteturas monolíticas, e com o desenvolvimento das grandes organizações, houve desafios que levaram as empresas a adotar uma arquitetura mais escalável para ter processos de desenvolvimento e de produção de soluções de software dos serviços mais sustentáveis [43].

Por muito tempo, empresas seguiram com arquitetura monolítica, que se baseia na criação de um único e grande executável em que toda a complexidade do software se concentra na mesma máquina, compartilhando recursos de processamento, memoria, base de dados e ficheiros. Ou seja, todas as funcionalidades e códigos encontram-se num único processo.

Hoje em dia, com um acréscimo de alta escala de aplicativos e serviços verifica-se a necessidade de garantir com o desenvolvimento e com o crescimento de aplicativos, uma adaptação ás exigências do mercado das tecnologias, e a cada vez mais exigências na qualidade do processo e no desenvolvimento dos serviços como resposta aos clientes de modo a segurar uma resposta aos requisitos com elevados níveis de fiabilidade, desempenho. Tornando assim cada vez mais complexo a manutenção e o gerenciamento desses aplicativos e serviços á medida que vão desenvolvendo.

A figura 1 faz a ilustração da arquitetura monolítica com a arquitetura dos microserviços, que dará ênfase para que a seguir se perceba a arquitetura de microserviços com o paradigma *service mesh.*

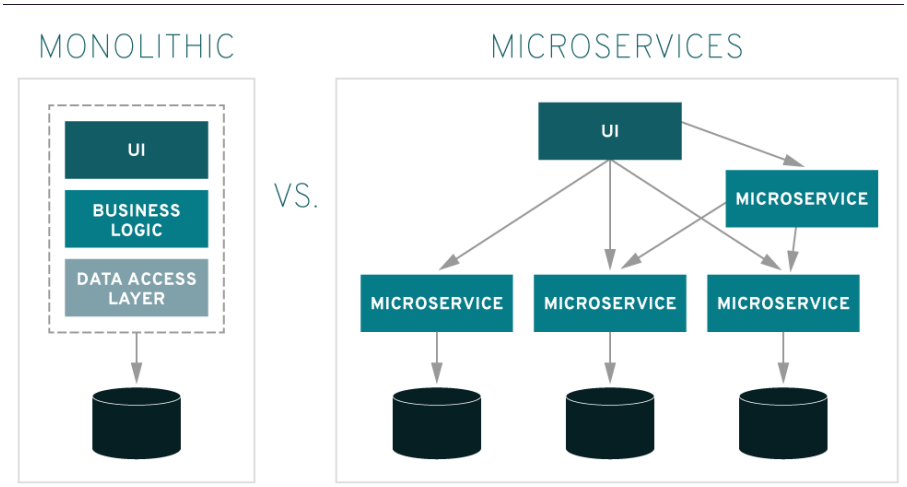

Figura 1: Ilustração da arquitetura monolítica e microserviços [13]

Com o aumento do desenvolvimento dos serviços, o modelo da arquitetura monolítica deixou de ser considerado uma melhor opção para grandes aplicações. À medida que a alteração de um único serviço, pode pôr em causa o funcionamento das suas dependências. Uma das maiores desvantagens e desafios da arquitetura monolítica, é que, em caso de uma falha, compromete o serviço todo.

Com isso houve a necessidade de empresas adotarem cada vez mais a uma arquitetura de microserviços, com a necessidade de aumentar a produtividade e agilidade dos aplicativos [3]. A ideia por trás da arquitetura de microserviços é tornar aplicações em serviços independentes que comunicam entre si, através de APIs e proporcionar agilidade no desenvolvimento dos serviços [16].
A arquitetura de microserviços fornece algumas vantagens como, independência e agilidade, ou seja, cada microserviço é totalmente independente e o *deploy* deles torna mais rápidos, assim como também é possível com a divisão dos serviços, trabalhar separadamente em cada um [3]. Uma grande vantagem da arquitetura de microserviços acima da arquitetura monolítica, é que, com a divisão dos serviços, os aplicativos tornam serviços independentes, e em caso de falhas, existe o suporte de manter o restante dos serviços em funcionamento. Portanto, a proposta da arquitetura orientada aos microserviços é desenvolver sistemas que sejam flexíveis, escaláveis e com uma manutenção mais simples do que as arquiteturas monolíticas.

Os problemas que acontecem numa arquitetura de microserviços, precisa ser resolvida de forma individual dentro dos *containers*, mantendo assim a integridade dos restantes serviços. Isso porque os containers encapsulam um aplicativo como um pacote de software executável que agrupa o código do aplicativo com todos os ficheiros de configuração, dependências e bibliotecas associadas que ele precisa para executar. Os aplicativos em container são isolados. Eliminando assim a sobrecarga de instalar e executar um sistema operacional em cada aplicativo, tornando os containers menores em capacidade (leve) e mais rápidos para iniciar, o que resulta em maior desempenho do servidor. Ao isolar aplicativos nos containers, reduz o risco de código malicioso afetar outros ou entrar no sistema.

Por isso que cada vez mais tem surgindo novas tecnologias que nos ajudam então a criar uma arquitetura resiliente, confiável, que seja mais fácil de ser gerenciado.

Criar uma arquitetura de microserviços escalável e resiliente é um desafio, porque na medida que o número de serviço vai aumentando, é preciso lidar com as interações entre serviços, monitora a integridade geral do sistema. Assim houve a necessidade de incluir a tecnologia *Service mesh*, nas arquiteturas de microserviços, facilitando a separação da rede lógica, permitindo que a concentração principal seja no aplicativo [9].
Os aplicativos do microserviços podem ser distribuídos em diversos servidores, *containers*, tornando-os serviços independentes. E a *service mesh* faz o gerenciamento do tráfego de rede entre esses serviços.

A *service mesh* foi desenvolvido a fim de proporcionar parte da carga operacional criada na arquitetura dos microserviços. A figura 2 mostra a ilustração da *service mesh* numa arquitetura de microserviços.

Assim, a *service mesh*, é uma abordagem de arquitetura para interligar microserviços e gerenciar o tráfego entre eles.

A *service mesh* é uma camada de infraestrutura que lida com a comunicação entre os serviços. E é responsável pela entrega fiável de solicitações por meio da complexa topologia de serviços que consiste em aplicativos nativos da nuvem. Na prática, a *service mesh* é implementada como uma matriz de proxies de redes, sendo implementados juntamente com o código do aplicativo [45].

A arquitetura da *service mesh* é composta por dois componentes de alto nível: o *control plane* e a *data control* [46]:

- O ***control plane*** é responsável pelo fluxo de dados do sistema, de modo que cada consulta seja processada pela data plane.
- Enquanto que a ***data plane*** é responsável por coletar e armazenar e supervisionar os proxies no plano de dados, fornecendo uma API que os interliga.

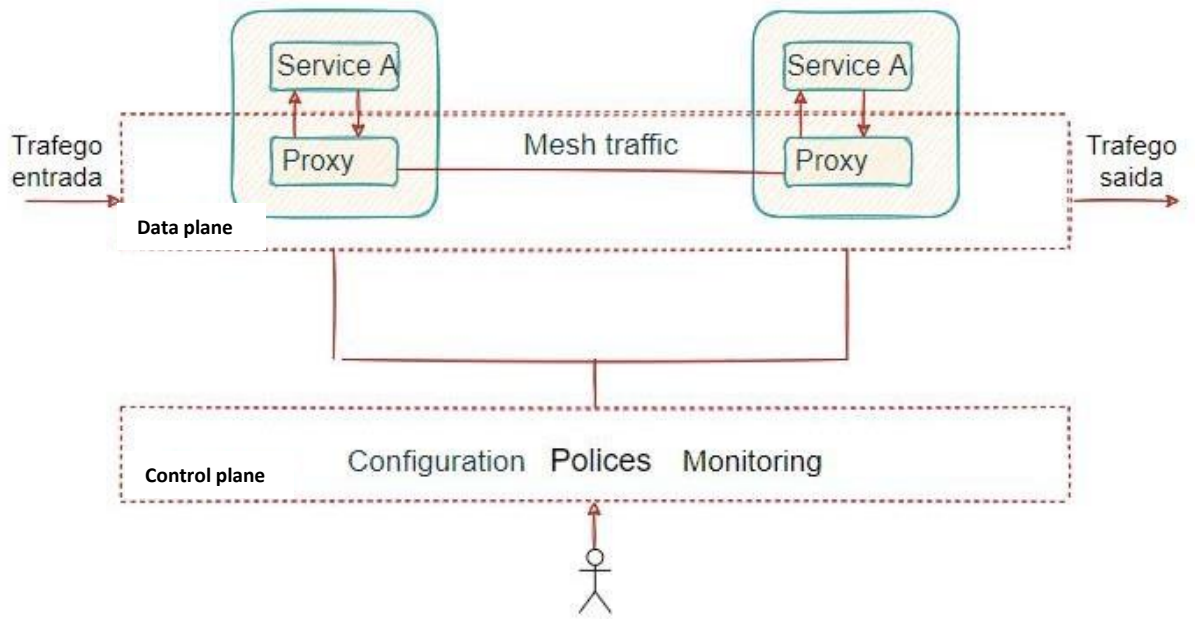

Figura 2: Arquitetura service mesh com microserviços

A *service mesh* é responsável por toda a comunicação feita entre os serviços, facilitando assim a carga dos desenvolvedores. De modo que os serviços não efetuam solicitações diretamente entre eles, toda a comunicação é feita por camadas, assim, os serviços são separados fazendo que as equipas possam implementar os seus serviços com mais autonomia.

Conforme é ilustrado na Figura 2, uma infraestrutura da *service mesh* é formado logicamente num plano de controle e um plano de dado.
A parte que faz o gerenciamento e controla todo o tráfego de rede entre as instâncias (Serviço A) é denominada de plano de dados, que vai lidar com toda a comunicação. Cada solicitação para a *service mesh* será processada pelo controle de dados.
O plano de dados é composto por um conjunto de proxies inteligentes que são normalmente implementados como *sidecars*. O plano de dados permite que a transferência de dados entre os utilizadores, por meio de protocolos, responsável assim pela descoberta de serviço, balanceamento de carga, autenticação, verificação de integridade e o monitoramento.

Enquanto que o plano de controle é projetado para conectar a uma API, fazendo a distribuição das configurações entre os proxies que estão no plano de dados.
Além de proporcionar políticas de configuração para execução de gerenciamento de tráfego, segurança e monitoramento. O plano de controle funciona como o cérebro da *service mesh.*

Quanto à segurança, é feito o gerenciamento de autenticação e autorização de serviços, além de oferecer suporte de encriptação com utilização de TLS mútuo.

Os proxies em *service mesh*, localizados no plano de dados, são conhecidos também de *sidecars*, porque são executados ao lado dos microserviços e não dentro deles, criando rotas de solicitações entre os serviços entre microserviços em sua respetiva camada de infraestrutura. Esses proxies controlam toda a comunicação da rede entre os microserviços, tendo assim uma visibilidade para cada pacote de rede.

2.6 Proxy de serviços

Na *service mesh*, realça-se também o conceito dos proxie de serviços, que são proxies do lado cliente para aplicativos de microserviços, que permite que os componentes do aplicativo enviem e recebam tráfego. Que funciona como um intermediário entre os componentes de um endpoint. O trabalho principal de um proxy de serviço é garantir que o tráfego seja encaminhado para o serviço correto de destino aplicando políticas de segurança, como também assegurar que as necessidades de funcionamento do aplicativo seja atendida [47].

O proxy reúne funções como:

- Controle de acesso;
- Filtro de conteúdo;

Com o controle de acesso, é possível determinar quem tem acesso à internet através de restrições aplicadas, oferecendo ao ambiente uma camada de proteção.
O Filtro de conteúdo determine também acesso autorizado pelo servidor, aplicando regras específicas a sites destinadas.

Quando a *service mesh* faz o uso dos proxies, são gerados proxies de entrada e proxies de saída. Proxies de entrada é todo o tráfego que entra, e proxies de saída é todo o tráfego que sai.

2.7 Características fundamentais da Service Mesh

Uma *service mesh*, é planificada para fornecer um conjunto de recursos fundamentais, conforme descrito a seguir [37] [38]:

- Descoberta de serviços

  O mecanismo de descoberta de serviço é importante, porque o número de instâncias assim como o estado e a localização de um serviço mudam dinamicamente ao longo do tempo. Assim, o mecanismo de descoberta de serviço, localiza um determinado serviço através de um pool que corresponde a uma instância de extremidade de descoberta, e envia uma solicitação para uma instância de serviço especifica, fazendo registo da latência e o tipo de resposta do resultado.

  Nesse caso a *service mesh* faz o uso da descoberta de serviços para escolher a instância com maior probabilidade de retornar uma resposta rápida com base da latência.

- Balanceamento de carga

  O mecanismo de balanceamento de carga oferece a capacidade de encaminhamento de tráfego pela rede, tendo em conta a latência e o estado das instâncias.
  O balanceamento de carga entre diferentes instâncias assegura que, se uma falhar, o resto do serviço possa continuar respondendo.

- Tolerância a falhas

  A *service mesh*, usa três estratégias para gerenciar a tolerância a falha: teste proativo, mitigação e teste de resposta rápida.
  Através desse conjunto de estratégias, são implementados processos e serviços afim de identificar falhas com uma certa antecedência. Assim como, implementar estratégias de mitigação de modo a minimizar o risco do impacto de qualquer tipo de falha.

- Monitoramento de tráfego

  Um outro recurso importante é o monitoramento de tráfego que oferece a taxa de sucesso relativamente á latência, volume das solicitações, assim como, o rastreamento distribuído e a capacidade de analisar a comunicação em tempo real serviço a serviço.

- Interrupção de circuito

  Interrupção de circuito é uma funcionalidade importante, capaz de interromper um circuito no caso de um acesso a um serviço sobrecarregado, do qual já contem uma alta latência, automaticamente ele recua as solicitações, em vez de falhar por completo um serviço com uma alta carga.

- Autenticação e controle de acesso

  A *service mesh* oferece mecanismos de segurança através do TLS, mTLS, e políticas de ACLS (lista de controle de acesso) que permitem atribuir políticas de controle que definem que serviços podem ser acessados e que tipo de tráfego não é autorizada e quais devem ser negados.

2.7.1 Principais vantagens da service mesh

A *service mesh* apresentam algumas vantagens como [43]:

- Faz com que o desenvolvimento seja acelerado, assim como testes, e implementações de aplicativos;
- Permite que mesmo pequenas empresas possam criar arquiteturas e funções que sejam altamente escaláveis;
- Balanceamento de carga;
- Suporte de recursos de configurações de segurança;
- Gerenciamento de tráfego e de encaminhamento;
- Permite o monitoramento;
- Permite mais rapidez e eficiência na atualização dos aplicativos;
- Os erros de comunicação são mais fáceis de identificar pois, sucedem dentro da própria camada de infraestrutura.
- Suporte de vários recursos de segurança, incluindo criptografia, autenticação e autorização;

2.7.2 Principais desvantagens da service mesh

- Alto nível de complexidade

  Um dos maiores desafios em torno da *service mesh*, é o alto nível de complexidade. Embora a *service mesh* tem o objetivo de simplificar a rede de comunicação entre os microserviços, mas para isso, em contrapartida, apresenta uma maior complexidade de

implementação, desde as terminologias de malha, os *proxies* e os *sidecars*, são recursos que exigem um pouco para ser mantidas pelas equipas.

- A comunicação inclui uma etapa adicional, a execução do serviço, primeiro precisa ser executada através de um *proxy sidecar*.
- Não oferece suporte de integração com outros sistemas ou sistemas.

2.8 Sumário

Neste capítulo, foram introduzidos conceitos e diferentes abordagens do *service mesh* que vão ser detalhadas e aprofundadas ao longo dessa dissertação.
Concretamente foi feito uma breve introdução do *service mesh*, nomeadamente os mecanismos de segurança utilizada, assim como alguns desafios. Foi feito também o levantamento e comparação entre as diferentes abordagens da *service mesh*, entre eles o istio, Kuma, linkerd, consult, etc.
Por fim foi abordado também a arquitetura da *service mesh*, juntamente com algumas das suas vantagens e desvantagens.
No próximo capítulo, é apresentado os aspetos relativamente á segurança da *service mesh.*

Capítulo 3

Segurança

3.1 Segurança do Service Mesh

A *service mesh*, surgiu como uma solução de gerenciamento e implementação de microserviços. Atualmente o desenvolvimento e integração de aplicações mais complexas utilizam a arquitetura de microserviços para implementar funcionalidades distintas.
Estas aplicações muitas vezes são implementadas em clusters de microserviços, tendo associada alguma complexidade em termos de manutenção, de monitorização e depuração.

Aplicações como a Netflix, Twitter, Spotify são exemplos de grandes serviços que optaram pela implementação de microserviços [25].
A Netflix é um dos mais populares provedores de serviço de stream online que foi uma das primeiras empresas a migrar com sucesso de uma arquitetura monolítica para uma arquitetura baseada em nuvem. Uma das razões que levou a Netflix migrar para a nuvem, foi devido ao grande e rápido aumento de dados e de informações de seus utilizadores, tornando assim uma tarefa um pouco difícil armazena-los em suas data center [58]. Assim, a Netflix com a *service mesh* introduziu vários *frameworks* de modo tornar seus serviços mais resilientes, sustentáveis e robustas.

A tendência do uso dos microserviços, tem-se vindo a tornar mais popular entre as empresas que procuram processos de desenvolvimento de aplicações e serviços mais ágeis.
Grande parte das empresas optaram primeiramente por projetar os seus serviços numa arquitetura monolítica, porém, com a arquitetura monolítica, toda a aplicação consiste em um único sistema. Para cada alteração no serviço é necessário executar um novo *deploy* de toda a aplicação, o que torna o processo bastante demorado e sujeito a indisponibilidade.

Com essa situação, grandes empresas optaram por microserviços, que em contraste à arquitetura monolítica, os microserviços, apresentam um sistema dividido em múltiplos serviços. A utilização dos microserviços, otimiza a produtividade assim como reduz o tempo de desenvolvimento, fazendo com que a implementação seja mais rápida. Tendo assim muitos benefícios e vantagens, por ter a aplicação dividida em diferentes serviços.

A arquitetura provém de um sistema onde cada aplicativo está dividido, do qual é mais fácil a manutenção e alterações rapidamente pelos desenvolvedores. Em contrapartida, do monolítico, os microserviços apresentam um alto nível de complexidade entre as comunicações entre os microserviços, assim lidar com falhas e latências na comunicação são dos desafios a enfrentar [25].

A *service mesh* pode ser utilizada para o balanceamento de carga, encaminhamento, monitoramento e segurança numa instância de um microserviços.
O tráfego da rede de um nó para outro, são emparelhados com proxies, que facilita a comunicação entre os serviços, permitindo assim, o controle e o monitoramento da rede.

Além da *service mesh* oferecer agilidade, escalabilidade, usabilidade e disponibilidade, os aplicativos baseados em *service mesh* apresentam alguns desafios, relacionados com a sua proteção. Isto porque os microserviços são serviços independentes sobretudo por serem sistemas distribuídos, do qual muitos estão expostas na Internet utilizando protocolos e padrões da web, como APIs REST e que são implementados dando prioridade a aspetos funcionais em detrimento de mecanismos de segurança:

- Com muitos microserviços há a necessidade de mais interconexões entre eles, assim como de proteção dos canais de comunicação.

- A natureza dos microserviços apresenta uma natureza granular, requerendo detalhes de autorizações em cada microserviços. Ou seja, isso faz com que seja preciso definir políticas de segurança para cada microserviços.

- Inexistência do conceito de perímetro de rede. Isto é, um perímetro de rede é conhecido por uma sub-rede física ou lógica que separa um conjunto de serviços numa rede.

3.2 Recomendações de práticas e implementações service mesh

A seguir estão descritas um conjunto de recomendações com diferentes aspetos que abordam a implementação e configuração de aplicativos de microserviços baseados em *service mesh*. Estas recomendações fazem parte de normas propostas pelo *NIST Special publication* 800-204A, *Building Secure Miroservices-based Applications Using Service-mesh Architecture* [48].

1 -Configurações de comunicações de proxies de serviços

- *Recomendação de tráfego permitido entre os proxies de serviços*
  Recomenda-se a existência de mecanismos capazes de determinar um conjunto de protocolos e portas em que o *proxy* de serviço está apto a aceitar o tráfego ao serviço associado. Tendo em consideração, que por padrão o *proxy* de serviço não deve permitir o tráfego.

- *Recomendação de acessibilidade entre os proxies de serviços*
  Recomendam-se recursos que limitem o acesso baseado em *namespace*, isto é, o acesso de controle da *service mesh* deve fornecer para a descoberta de transmissão, políticasdiferentes e dados de telemetria.

- *Recomendações de recursos de tradução de protocolos*

Os proxies de serviços devem ter recursos adaptados para suportar que clientes comuniquem com diferentes protocolos dos microserviços.
Essa tradução de protocolos permite evitar a exigência de construir um servidor distante por protocolo de cliente, que por consequência, aumentaria zona de ataques.

2 - Recomendações de proxies de entrada

- *Configurações de proxies de entrada*
  Recomendam-se recursos que permitem configurar o tráfego e regras de encaminhamento dos proxies de entrada, e os proxies de serviço.

3 - Recomendações de configurações de acesso a serviços externos

Alguns serviços baseados em microserviços podem ter a necessidade de aceder a algum serviço externo, APIs de web privadas, ou até mesmo, aplicativos em *clusters* diferentes.
Para o nível de segurança, são recomendados os seguintes:

- *Recomendações de restrição de acesso a recursos externos*
  Devem ser desabilitados por padrão, todos os acessos a recursos externos, fora da *service mesh*, e permitida apenas com uma política explícita.

- *Recomendações de proxies de saída*
  Disponibilidade de recursos de configurações que incluem regras de encaminhamento do tráfego para proxies de saída, assim como os de entrada.

- *Recomendações de acesso a recursos externos*
  Disponibilidade de recursos de melhoria, sendo configuradas dentro da *service mesh* para fornecer acesso a recursos externos e serviços.

4 - Configurações de gerenciamento de identidade e acesso

A comunicação de um aplicativo baseado em microserviços é constituído por três entidades principais, *clientes, microserviço e serviços externos.*

Recomendações implementadas em certificados:

- *Domínio de identidade universal*
  As identidades de um microserviço deve ser consistente e exclusive, de modo que, o serviço deve ter a mesma identidade independente de onde está localizado, e deve ser único no sistema.

- *Implantação de certificado de assinatura*

- *Rotação de certificado de identidade*

  O certificado da identidade de um microserviço tem de ser curto tanto quanto o gerenciamento dentro de uma infraestrutura, isso ajuda a limitar ataques.

- *Ciclo de conexões na mudança de identidade com service proxy*
  Após alterações de um certificado de identidade de um *proxy* de serviço, este deve ser desativado com eficácia, e estabelecer todas as novas conexões com um novo certificado. Para evitar ataques no tempo, os certificados são validados apenas durante o *handshake* mTLS, em que as conexões são substituídas, assim que um novo certificado é emitido.

- *Autenticação de carga de trabalho antes da emissão do certificado*
  Antes de emitir o certificado de identidade, o sistema que gerência os certificados do plano de controlo do *service mesh* tem que proceder à autenticação dos serviços.

- *Serviço seguro de nomes*
  Recomenda-se que o *service mesh* proporcione um serviço de nomes seguro que contém o mapeamento e a identidade do servidor que contém o nome do microserviço que é fornecido via DNS, isto quando o certificado for utilizado por mTLS, pois ele carrega as informações do servidor.

- *Identificação granular*
  Cada microserviço do *service mesh* deve apresentar identidades próprias.
  Como cada serviço com a sua própria identidade, é possível ter acesso a um microserviço específico.

- *Escopo de política de autenticação*
  A política de autenticação deve especificar a política de autenticação de cada serviço.

- *Token de autenticação*
  A autenticação através de *token*, deve ser assinada digitalmente e mecanismos criptográficos fortes a fim de garantir autenticidade.

5 - Configurações de capacidade de monitoramento

Os dados de monitoramento devem ser recolhidos para todos os serviços *proxy*, tanto de entrada e saída.

- *Recomendações para eventos de registros*
  Aos eventos de registros, os serviços do proxy devem ser capazes de registar todos os erros que podem surgir na validação de entrada.

- *Solicitações de registros*
  O proxy tem de fazer o registo de pelo menos dos campos *Common log Formant* (formato de arquivo de texto padronizado utilizado por servidores da web ao gerir logs de servidor) para as solicitações irregulares.

- *Conteúdo de mensagem de log*
  Ao apresentar informações como, data, hora, identidade do microserviço, no log fica mais acessível o rastreamento, em caso de monitoramento de um microserviços específico.

- *Métricas obrigatórias*
  O *service mesh* deve manter a configuração que coleta as métricas para comunicações externas de clientes e microserviços, tendo em conta, o número de solicitações feito entre o cliente/serviço em um determinado período, assim como informações sobre o número de falhas dessas solicitações.

- *Implementação de monitoramento distribuído*
  A implementação de monitoramento distribuído é importante no momento da monitorização. Ao fazer a configuração dos proxies é importante para a implementação distribuído, assegurar que os serviços encaminhados contenham a informação dos pacotes de comunicação.

3.3 Melhoria da disponibilidade por meio de técnicas de resiliência da rede

A seguir, de acordo com a documentação da NIST, está disponível um conjunto de recursosque ajudam na melhoria da disponibilidade, afim de melhorar a resiliência da rede de comunicação entre os serviços.

- **Balanceamento de carga**
  Para evitar atrasos nas respostas, ou em serviços devido a sobrecargas, é necessário ter várias instâncias do mesmo serviço e a carga dessas mesmas instâncias devem ser distribuídas de forma uniforme.

- **Disjuntor (*Circuit breaker*)**
  Disjuntor é a política utilizada para procurar erros no tráfego da comunicação, isto é, em sistemas distribuídos em grande escala. Independente de como foram arquitetados, eles fornecem alta probabilidade de pequenas falhas, portanto, a malha de serviço deve ser projetada para proteger dessas situações, através de redução de cargas.
  Isto é, quando houver uma falha de uma instância dos microserviços, interrompendo o encaminhamento das solicitações para essa instância, deve envolver um limite de resposta. Reduzindo assim, a possibilidade de falhas em partes, permitindo assim, tempo para análise de registos, correções de falhas.

- **Limitação de taxa**
  De forma a garantir disponibilidade contínua do serviço, as taxas de solicitações que chegam ao microserviços têm de ser limitadas.

- **Monitorização**
  A monitorização é importante, porque permite detetar ataques e identificar fatores que podem corromper a comunicação entre serviços, impactando a disponibilidade. Portanto, é de necessidade monitorar o tráfego de rede dentro e fora, através de rastreamento, geração de métricas, desempenho de análise.

3.4 Recursos de resiliência/ estabilidade para comunicação

A seguir apresenta alguns dos recursos de resiliência da *service mesh* para a comunicação, nomeadamente: disjuntores, novas tentativas, tempo limite, injeção de falha e balanceamento de carga.

- **Controlador de entrada**
  O proxy de serviço de uma malha de serviço pode ser implementado para controlar o tráfego de entrada, através de algumas funções como:
  - Balanceamento de carga;
  - Encerramento de TLS publico;
  - Tradução de protolocos seguros;

- **Controlador de saída**
  O proxy de serviço de uma malha de serviço pode ser implementado para controlar o tráfego de saída, como um proxy sidecar através de algumas funções como:
  - Troca de credenciais, ou seja, conversão de credenciais de identidade interna para a externa;
  - Tradução de protocolos;

- **Em nível de comunicação entre os proxies de serviço**
  <u>- Permissão de tráfego entre proxies</u>
  deve haver um recurso para especificar um conjunto de protocolos e portas em um proxy de serviço para aceitar tráfego para seu serviço associado.

  <u>- Acessibilidade de proxies</u>
  Deve haver um recurso para limitar o acesso;

  <u>- Tradução de protocolo</u>
  Deve haver um recurso integrado que oferece suporte a clientes que se comunicam com protocolos diferentes.

(*Isso é necessário para evitar a necessidade de construir um servidor separado por protocolo de cliente, o que aumenta a superfície de ataque*.)

- Proxie de entrada
Deve haver recurso de configuração de regras de encaminhamento para cada proxie.

- **Em nível de gerenciamento de identidade de acesso**
  As entidades precisam ter identidades distinta e executar autenticação mútua.

- **Em nível de implementação de certificados**
  - Recomendação de um domínio de identidade universal
  Isto é, a identidade de todas as instâncias de um microserviços deve ser consistente e exclusivo.

  - Implementação de certificado de assinatura
  O sistema de gerenciamento de certificados do plano de controle do *Service Mesh* deve ter a sua capacidade de gerar certificados auto assinados desabilitada.

  Em vez disso, o certificado de assinatura utilizado pela malha plano de controle deve sempre estar enraizado na raiz de confiança da PKI existente da empresa e fornecido com segurança ao plano de controle da *Service Mesh* na inicialização.

  - Rotação de um certificado de identidade
  A vida útil de um certificado de identidade deve ser tão curto quanto o gerenciamento na infraestrutura.
  (Isso ajuda a limitar os ataques, pois um invasor só pode utilizar uma credencial para personificar um serviço até que a credencial expire e roube novamente uma credencial com sucesso, aumentando a dificuldade para um atacante).

- **Em nível de monitoramento**
  Todos os proxies (entrada e saída) devem ter a capacidade de coletar todos os dados de monitoramento.

- **A nível de técnicas de resiliência da rede**
  - Implementação de dado para implementação de resiliência de rede
  Aplicar recursos de tempo limite, configuração de interrupção de circuito.

  - Implementação de verificação de integridade de instâncias de serviçoAplicar recursos de verificação de integridade.

  Recursos como, função de disjuntor, balanceamento de carga, e limitação de taxa, são utilizados para a necessidade de melhorar a disponibilidade e resiliência dos serviços.

- **A nível de configuração de registo de serviços**
  - A comunicação entre os serviços deve ocorrer por meio do protocolo de comunicação seguro, como HTTPs ou TLS.

  - O registo de serviço deve ter verificação de validação afim de garantir que apenas os serviços legítimos estejam realizando operações e consultas na base de dados.

- **Estratégias de segurança para comunicação segura**
  -Os clientes não devem ser configurados para chamar os serviços de destino diretamente, mas sim para apontar para a URL do gateway único.

  - O gateway de cliente para API, bem como a comunicação de serviço a serviço, deve ocorrer após a autenticação mútua e ser criptografada.

  - Os serviços que interagem com frequência devem criar conexões TLS de manutenção de atividade.

3.5 Sumário

Neste capítulo, foi abordado aspetos de segurança importante que contribuem para o desenvolvimento desse trabalho, que é a segurança na comunicação entre microserviços em uma framework do Kuma.
Foi feito o levantamento do estudo e analise de um conjunto de recomendações de segurança de acordo com a documentação da NIST, com diferentes aspetos que abordam a implementação e configuração de aplicativos de microserviços baseados em *service mesh.*
Com esses conjuntos de recomendações, permite-nos uma orientação de configurações e implementações detalhadas para cada um dos componentes do *service mesh.*

Capítulo 4

Análise da framework Kuma

Neste capítulo, será documentado o estudo da análise detalhada do Kuma que inclui o seu funcionamento, suas dependências, políticas e gestão dos certificados de modo a avaliar a performance dos serviços com a *service mesh* através da framework Kuma.

4.1 Framework do Kuma

A *framework* do Kuma, é um plano de controle de código aberto universal para *Service Mesh*, e microserviços que oferece suporte *envoy* como uma tecnologia proxy de *dataplane* (planos de dados). Embora que não requer uma experiência em *envoy*, isto é, o Kuma é capaz de abstrair as políticas que iremos utilizar, e caso exista uma política que o Kuma não tenha suporte nativamente, pode-se utilizar a política de modelo de proxy com as configurações do *envoy*.
De modo que, se houver algo que o *envoy* pode fazer, e o Kuma não, ainda assim teremos acesso ao ecossistema do *envoy* através da configuração do modelo proxy [20].

A *framework* do Kuma nativo do kubernetes, VMs e Multi-Mesh, suporta vários serviços em um cluster. O Kuma fornece conectividade multizona mais escalável em vários clusters em kubernetes, VM ou híbridos [4].

Como descrito anteriormente, o Kuma é arquitetado para funcionar em kubernetes e VMs. O Kuma atualmente é um projeto sandbox da CNCF (cloud native computing), projetado para reduzir a complexidade de execução numa malha de serviço com vários recursos como suporte de várias zonas em muitos clusters, como data centres e nuvens diferentes.
De modo geral, o Kuma apresenta um plano de controle de código aberto, que gerência malhas de serviços e microserviços com suporte a ambientes kubernetes e VM.

O Kuma suporta implementações e abordagens de malhas de serviços em implementações distribuídas.
Isto é, o Kuma é:

- *Universal e nativo do Kubernetes,* independente, ou seja, capacidade de ser executado em qualquer lugar;
- *Autónomo e multi zona,* capacidade de várias zonas com suporte em descoberta deserviços;
- *Multimalha*, suporte de várias malhas individuais;
- *Políticas baseadas em atributos,* suporte de uma vasta gama de políticas e recursos;
- *Escalável*;

No Kuma, numa implementação de *multi zona*, existem recursos importantes que o Kuma suporta:

- Existência de dois modos de plano controle: o global e o remoto;
- Suporte da zona DNS, para descoberta de serviços;
- Um tipo de proxy de dados denominados como ‘’ingress’’ que permite a ligação entre aszonas dentro de uma malha no Kuma;

Em uma aplicação distribuída no Kuma, onde engloba os dois modos de plano de controle, o modo global, é encarregado de aceitar recursos do Kuma afim de determinar o comportamento da malha de serviço por meio de kubernetes nativos ou YAML (arquivo de configuração, em formato de serialização de dados legíveis por humanos) em implementações baseadas em VM’s, responsável por difundir esses recursos para o plano de controle ‘’remoto’’, um para cada zona do qual queremos, enquanto que o plano de controle ‘’remoto’’ troca recursos do Kuma com o plano de controle ‘’Global’’ através da extensão de uma API Envoy denominado de KDS (Kuma Discovery Service) por um protocolo GRPC e também por HTPP/2.

O Kuma tem um plano de controle (Kuma-cp) juntamente com o Envoy que é um proxy distribuído de alta performance planificada para serviços e aplicativos únicos, assim como a interligação de comunicação entre os dispositivos como o plano de dados universal projetada para grandes arquiteturas de *service mesh*.
Construído acima de soluções como HAproxy, um software de código aberto que fornece balanceamento de carga e servidor proxy de alta performance para aplicativos baseados em TCP/HTTP, espalhando solicitações entre os vários servidores [52]. Assim como balanceadores de carga na nuvem, o envoy é executado juntamente com todos os aplicativos, abstraindo a rede fornecendo recursos comuns de forma independente de plataforma, de modo que quando todo o tráfego de serviço numa infraestrutura flui por meio de uma malha do envoy, facilitando a visualização de áreas problemáticas por meio de absorvibilidade continuo de forma a ajustar o desempenho geral e adição de recursos num único local [2].

O envoy engloba recursos como:

- Support HTTP/2 e GRPC para conexões de entrada e saída.
- APIs de gerenciamento de configurações, o envoy fornece APIs robustas para gerenciar as configurações de uma forma dinâmica.
- Absorvibilidade, o envoy suporta absorvibilidade profunda através da camada L7 (Camada de aplicativo, onde são aplicadas as políticas e regras), com rastreamento distribuído.
- Balanceamento de carga avançado, suporte de recursos avançadas de balanceamento de carga, incluindo interrupção de circuito, limitação de taxa, incluindo novas tentativas automáticas de conexão e serviços.

Perante este conjunto de recursos suportado pelo envoy, o Kuma por si já abstrai a complexidade do envoy agrupando-o no (Kuma-dp) de modo que o utilizador não precisa de

configurar diretamente o envoy, tornando assim, o processo de início de um proxy de plano de dados mais fácil.

Ao impulsar API xDS (Serviço de descoberta) nativa do Envoy que conecta com o Kuma-dp com plano de controle ‘’remoto’’ perante todas as zonas, assim como a conexão do KDS (Kuma Discovery Service) com o plano de controle ‘’remoto’’ ao plano de controle global, com a comunicação GRPC eficiente em toda a pilha de infraestrutura de malha de serviço de forma resistente.

Existência de dois modos de plano controle: o global e o remoto tem alguns benefícios como:

- Podemos avaliar a cada zona independente dimensionando o plano de controle ‘’remoto’’ e obter alta disponibilidade e resiliência em caso de problema em uma zona.

- O plano de controle global permite-nos automaticamente generalizar o estado em todas as zonas ao mesmo tempo, de modo que possamos certificar de que planos de controle ‘’remotos’’ estão cientes da entrada do Kuma em cada zona habilitada a ligação entre zonas.

- Não existe um único ponto de falha, isto porque, mesmo que o plano de controle global caia, ainda podemos registar e cancelar o registo dos proxies de plano de controle remoto, onde os endereços de cada serviço mais atualizado ainda podem ser distribuídos para outros *envoys sidecars.*

4.2 Funcionamento do Kuma

Para construir qualquer aplicativo, inevitavelmente introduzimos serviços que vão se comunicar entre si através de solicitações na rede.

Exemplo de um cenário:
Em um cenário em que temos um aplicativo que comunica com uma base de dados para armazenar ou recuperar dados, ou ate mesmo uma aplicação de microserviços mais complexo que tenha necessidade de executar muitas solicitações entre os diferentes serviços, toda vez que esse serviço se comunica com a rede, coloca em risco o tráfego com o utilizador, isso porque a rede entre diferentes serviços pode ser lenta e imprevisível, apresentando problemas como: controle de versão, encaminhamento).

E é com isso que o Kuma apresenta ações que são capazes de mitigar essas situações através do proxy sidecar.

O Proxy Sidecar, é um proxy secundário que executa ao lado do processo do serviço no mesmo *host* subjacente. Ou seja, o sidecar proxy abstrai alguns recursos como comunicações entre serviços, monitoramento e segurança, longe da arquitetura principal para facilitar o rastreamento e a manutenção do aplicativo como um todo. Normalmente são aplicados no plano

de controle da *service mesh.*

Com o proxy sidecar os serviços transferem todas as questões de conectividade e monitoramento para um tempo de execução, do qual faz conexão e estará no caminho de execução de cada solicitação. Fazendo proxy de todas as conexões de saída e aceitando todas as conexões de entrada, executando as políticas de tráfego em tempo de execução como o encaminhamento. Com isso os desenvolvedores não precisam de preocupar com a conectividade, concentrando assim nos seus serviços e aplicativos.

O Kuma tem uma instância de um proxy sidecar para cada instância em execução com os serviços, enquanto que todas as solicitações de entrada/saída passam sempre pelo proxy sidecar. O número de instância para os serviços coincide ao número de proxies sidecar, isto é, temos um proxy sidecar para cada microserviços.
O proxy sidecar requer um plano de controle do qual permite que seja possível a configuração do comportamento dos proxies dinamicamente.
O Kuma, usa como objetivo principal, reduzir o código que precisa ser escrito e mantido afim de criar arquiteturas robustas.

Portanto, o Kuma adota o modelo de proxy sidecar beneficiando do envoy com a sua tecnologiade plano de dados sidecar.

Ao abstrair das preocupações de conectividade, segurança e encaminhamento para um sidecar proxy, o Kuma beneficia dos seguintes:

- Capacidade de criação de aplicativos mais rápidos;
- Ao terceirizar as preocupações de conetividade, encaminhamento e segurança para o sidecar proxy, o foco centra-se na funcionalidade principal nos serviços;
- Construção de uma arquitetura mais segura, afim de reduzir a fragmentação;

4.3 Dependências do Kuma

O Kuma controle plane (Kuma-cp) é um executável em GoLang (uma linguagem de programação criada pela google), pode ser instalada em qualquer lugar, por isso que é universal e fácil de implementar.

O Kuma pode ser implementado em:

- Kubernetes, em kubernetes não é preciso dependências externas, por causa do proveito do servidor do API K8s, nesse caso o Kuma injeta automaticamente proxies do plano de dados sidecar.

- Universal, em modo universal o Kuma precisa de uma base de dados PostgreSQL, com dependências para armazenar configurações.

Os Kubernetes, conhecido também pela abreviação de K8’s orquestra aplicativos em contêineres afim de serem executados em cluster de hosts, isto é, automiza a implementação e gerenciamento de aplicativos nativos de nuvem utilizando infraestruturas locais ou em plataformas de nuvem publica.

4.4 Políticas suportadas pelo Kuma

O objetivo desta sessão é apresentar aos conjuntos de recursos e políticas de configurações para um funcionamento melhor e mais seguro para uma melhor funcionalidade dos serviços. Aos conjuntos de políticas suportadas pelo Kuma temos:

- Políticas e recursos a nível de Segurança temos:
  - Multi-Mesh (*multi-mesh)*
  - Mutual TLS (*Mútuo TLS)*
  - Traffic Permissions (*Permissões de tráfego)*

- Políticas e recursos a nível de Tráfego de controle temos:
  - Traffic Route (*Rota de trânsito)*
  - Health check; *(Exame de saúde)*
  - Circuit breaker *(Disjuntor)*
  - Fault injection *(Injeção de falhas)*
  - Kong Gateway *(Kong gateway)*
  - External Services *(Serviços externos)*
  - Retries *(novas tentativas)*
  - Timeouts *(tempo limite)*
  - Rate limite *(taxa de limite)*
  - Virtual Outbound *(saída virtual)*

- Políticas e recursos a nível de Observability:
  -Traffics Metrics (*Metricas de tráfego*)
  - Service Map (*Mapa de serviço*)
  - Traffic trace (*Rastreamento de tráfego*)
  - Traffic Log (*Registro de tráfego*)

- Advanced
  - Proxy template (*modelo de proxy*)
  - DP/CP Security (*Segurança DP/CP*)

O Kuma apresenta um conjunto de políticas, recursos e configurações que ajudam a melhorar a segurança.

- **Multi-Mesh (*multi-mesh)***

O multimesh é um recurso de muita importância no Kuma, basicamente o multimesh é a capacidade de criar várias malhas de serviço isoladas dentro de um mesmo cluster no Kuma, o que dá ao Kuma a facilidade de operar em ambientes que seja necessário mais que uma malha.

O mesh no Kuma, é um recurso que inclui: **Proxies do plano de dados e políticas**
A utilização de pelo menos um mesh no Kuma deve existir, e não tem limite de número de malha a serem criadas no Kuma.

O proxy de plano de dados que conecta com o plano de controle que fica localizado no Kuma-cp, especifica em qual mesh pertence. Enquanto que um proxy de planos de dados pode apenas pertencer a um mesh por vez.

Nota: **Sempre que seja iniciado um novo cluster do zero, é criado automaticamente um mesh por defeito.*

Alem da capacidade de criação de uma mesh de serviço virtual, é utilizado também recursos de:

- TLS mútuo
- Métricas de tráfego
- Traffic trace

O *TLS mútuo* serve para proteger e criptografar o tráfego de comunicação entre os serviços, assim como atribuir uma identidade aos proxys do plano de dados dentro da mesh.

As *Métricas de tráfego*, é responsável pelas configurações do back-end de métricas que é utilizado para a recolha e visualização das métricas de tráfego na malha de serviço.

O *Traffic Trace***,** é o recurso utilizado para configurar o back-end de rastreamento que irá ser utilizado na recolha de rastreamento de tráfego do serviço que esta sendo executado dentro da mesh.

- **Mutual TLS (*Mútuo TLS)***

O Mutual TLS é uma política do Kuma que permite que o tráfego seja criptografado automático e também fornece uma identidade para cada proxy do plano de dados para todos os serviços em um mesh. O Kuma faz o gerenciamento de um certificado para cada proxy de plano de dado de um ficheiro mesh.
Para habilitarmos o recurso do mTLS, é preciso configurar o mTLS nas propriedades do mesh. Sendo que o Kuma suporta *back-ends*, podemos ter quanto quisermos, porém, apenas um de cada vez pode ser habilitado através da propriedade denominada de *enableBackend.*

Se o recurso *enableBackend* estiver ausente ou até mesmo vazio, o mTLS fica inativo em toda a malha.
O Kuma fornece também *back-ends* que suportam CA (certificate authority), assim como rotações automáticas de certificados.

Portanto, o Kuma fornece as seguintes *back-ends* suportadas pelo CA (certificate authority): modo *builtin*, e modo *provided.*

- O modo *Builtin*, faz o gerenciamento automático de um CA e uma chave raiz CA, que vão ser armazenados automaticamente como um secret, que é um recurso definido pelo Kuma, pertencente a um mesh, é um recurso específico que não deve ser partilhado entre os diferentes mesh.
Com o *builtin*, o Kuma faz o gerenciamento dinâmico do seu próprio certificado raiz do CA (certificate authority), e da chave que vai ser utilizado para supervisionar certificados automaticamente para cada réplica de cada serviço.
É possível especificar mais de um *builtin* com diferentes nomes, e cada um poderá automaticamente provido com um único par de certificado.

Os *secrets* do Kuma são armazenados no mesmo *nameSpace* (Um termo utilizado para delimitar um containers que fornece um contexto de itens onde armazena nomes, termos, técnicas) [53], que o controle plane no *system.Kuma.io/secret*
Os *secrets* do kubernetes são identificados com *namespace* como dito anteriormente, de modo que não seja possível ter um *secret* com o mesmo nome em várias malhas de serviço, mesmo que várias malhas pertencem ao Kuma-cp, executando em um *namespace*.

O Secret é um recurso suportado pelo Kuma que mantêm as informações confidenciais armazenadas através de uma interface.

- O modo *Provided***,** faz com que o certificado raiz da CA (certificate authority), e a chavefornecidos pelo utilizador estejam no modo *secret*.
Os certificados gerados no modo *provided*, é feito através do fornecimento da chave raiz do CA (certificate authority), pelo utilizador, com essa opção o Kuma provisiona o certificado do proxy de plano de dado para cada réplica de cada serviço do certificado raiz e da chave da CA.

Após a especificação de um *back-end*, o Kuma faz o gerenciamento automático de um certificado para cada proxy de plano de dados no ficheiro mesh.
Os certificados que o Kuma gerência são compatíveis com o SPRIFEE (é um plano de controle de identidade universal para sistemas distribuídos) [54], utilizado em casos de uso de autenticação de identidade afim de identificar cada carga de trabalho no serviço.

Os certificados que o Kuma gera utiliza o SAM (Storage área network), que basicamente é uma rede de dispositivos de armazenamento de dados em rede [55].
O Kuma quando aplica políticas que exigem identidade como no recurso *TrafficPermission*, extrai o SAM do certificado do cliente que posteriormente corresponde à identidade do serviço.
Lembrando que por padrão o recurso mTLS estão desabilitados, portanto, assim que o mTLS for habilitado, todo o tráfego é negado, a menos que o *TrafficPermission* seja configurado para permitir a comunicação entre os proxies.
É importante certificar sempre que o recurso *TrafficPermission* esteja presente antes de habilitar o recurso do mTLS em uma malha de serviço, afim de evitar interrupções no tráfego causado pela falta de autorização entre os proxies.

O mutual Tls quando habilitado no modo *builtin*, cada mesh fornecerá o seu próprio certificado e chave de CA (certification authority), a menos que seja explícito o uso da mesma CA (certificate authority), partilhado entre os diversos malhas.
Acontece que quando o CA (certificate authority), que está nas malhas é diferente, os proxys de plano de dado de uma mesh não podem consumir proxys de plano de dados que pertecem a outro mesh, portanto é preciso uma API intermedio habilitado para fazer a comunicação entre asmalhas.

No Kuma, qualquer certificado TLS que vai ser emitido pelo plano de controle deve ter uma CA (certificate authority), que pode ser gerada automaticamente pelo Kuma por meio de um *back-end* integrado, ou então através da inicialização com um certificado e chave do cliente através de um *back-end* fornecido.

Os certificados do proxy de dados são gerados automaticamente pelo Kuma, certificados X.509 do tipo SPIFFE[1], que estabelece identidades de serviço, uma estrutura de certificado compatível, um conjunto de padrões de código aberto utilizado para identificar com segurança sistemas de softwares tanto em ambientes dinâmicos como em ambientes heterogéneos.
O Kuma permite que possamos escolher uma autoridade de certificado diferente, ou seja, o Kuma fornece uma CA integrada assim como nos permite a possibilidade de optar por fornecer a nossa própria CA.

- **Traffic Permissions (*Permissões de tráfego)***

O traffic permissions é uma política utilizado pelo Kuma que fornece regras de controle de acesso para definir o tráfego da comunicação dentro da *mesh*.

[1] https://spiffe.io/

Para que as permissões do tráfego permitam o tráfico, é preciso que o TLS mútuo seja habilitado, isso porque o Kuma precisa avaliar a identidade do serviço com os certificados de proxy do plano de dados.
Ou seja, quando o TLS mútuo está desabilitado, o Kuma permite o tráfico para todo o serviço. Por padrão, o *traffic permissions* que o Kuma cria quando é instalado, permite toda a comunicação entre os serviços. Por isso, é preciso configurar juntamente com o mTLS mútuo, as configurações apropriadas de acordo com a necessidade de cada serviço em malha.

O Kuma apresenta um conjunto de políticas, recursos e configurações que ajudam a melhorar o controle de tráfego.

- **Traffic Route (*Rota de trânsito)***

O traffic route é uma política que permite configurações de encaminhamento do tráfego dentro da malha de serviços, oferecendo capacidade de encaminhamento.

As configurações devem ser explícitas no proxy de planos de dados definindo as regras de encaminhamento.

O Kuma também oferece suporte de balanceamento de carga, e reconhecimento de localidade do serviço, o que é bastante importante em uma implementação de *multizonas,* porque através disso, o balanceamento de carga comunica com os proxys de plano de dados na tentativa de manter as solicitações em uma zona. Ou seja, a quantidade do tráfego que vai permanecer em uma determinada zona, depende da integridade dos pontos de extremidades de serviço implementados nessa zona.

O plano de controle do Kuma cria por padrão um *trafficRoute* toda a vez que for criado um meshnovo. E por padrão o *trafficRoute* habilita todo o tráfego entre os serviços.

- **Health check; *(Exame de saúde)***

O helth check é uma política que permite que o Kuma tenha a capacidade de acompanhamento da integridade de cada proxy de plano de dados, afim de minimizar o número de solicitações com falhas no caso de um proxy de plano de dados fique temporariamente íntegro.
Este recurso, instrui ao proxy de plano de dados o acompanhamento do estado da integridade de qualquer outro proxy de plano de dados, com objetivo de verificar a configuração da integridade está corretamente. Um proxy de plano de dados jamais irá enviar solicitações para outro proxy sem a verificação da integridade da mesma.
O Kuma efetua o envio de solicitações entre os serviços, somente após a verificação do estado integro.

- **Circuit breaker *(Dijuntor)***

O circuit breaker é a política utilizada para procurar erros no tráfego da comunicação que são trocados entre os proxys de plano de dados em tempo real, de modo a sinalizar um proxy de plano de dados não íntegro, de modo a garantir que nenhum tráfego adicional atinja o plano de dados não íntegro até que esteja integro novamente.
O recurso *circuit break* ao contrário do *health check* que verifique a integridade ativa, o *circuit break* não envia tráfego adicional para os proxys de planos de dados, porém analisa a existência do tráfego de serviços.

Aos recursos de *circuit breaker*, ele apresente 5 condições que determine o estado do ‘’disjuntor’’:
1-Total de erros;
2-Erros de gateway;
3-Erros locais;
4-Desvio padrão;
5-Falhas;

- **Fault injection *(Injeção de falhas)***

O fault injection é a política utilizada pelo Kuma com a finalidade de testar a resistência dos microserviços.
O Kuma apresenta três tipos de falhas que podem ser encontradas no seu ambiente.
1- Delay;
2-Abort;
3-ResponseBandwidth;

1-Delay(atraso), representa a configuração de atrasar uma resposta de um destino.

2-Abort, representa a configuração de não entregar as solicitações ao serviço do destino, e sim, fazer a substituição das respostas no plano de controle de dados de destino através do código predefinido.

3- ResponseBandwidth, define configurações de limites de largura de banda de resposta das solicitações.
O *ResponseBandwidth* apresenta dois parâmetros:
- *limit*- representa a medida de valor em gbps, mbps, kbps.
- *percentage*- é a percentagem de solicitações do qual o limite de largura de banda da respostaserá injetado. Esse intervalo de percentagem vai de [0,0 – 100,0].

- **Kong Gateway *(Kong gateway)***

O Kong Gateway é um recurso do Kuma que faz a rota do tráfego da rede de fora de uma mesh do Kuma para serviços dentro da mesh.
O kong gateway é implementado como um Kuma *dataplane*, ou seja, na instância do Kuma-dp, onde faz o gerenciamento de um processo de proxy envoy que faz o proxy de tráfego de rede real.

Neste contexto existem dois tipos de gateways:
- O *Delegated*, que é o que vai permitir que os utilizadores usem qualquer gateway existente como o kong.
O kong ou kong APi gateway é uma APi nativo independente e escalável, alto desempenho e extensibilidade, fornecendo funcionalidades para proxy como, encaminhamento, balanceamento de carga e verificação de integridade.

- O *Builtin*, é o recurso que configura o proxy do plano de dados que faz a exposição externos para direcionar o tráfego para dentro da mesh.

**Nota*: Os gateways existem dentro de uma malha. Em caso de várias malhas, é preciso que cada malha tenha o seu próprio gateway.

- **External Services *(Serviços externos)***

External Services é uma política suportado pelo Kuma que permite que os serviços executados dentro de uma determinada malha possam consumir serviços que não fazem parte desta malha, como, por exemplo o mesmo monitoramento, segurança, encaminhamento do tráfego para tráfego externo como, para serviços que fazem parte da malha.

- **Retries *(novas tentativas)***

Retries é a política suportada pelo Kuma, que faz com que o Kuma saiba como comportar em casos de falhas. Como por ex.: em situações de repetições de solicitação de HTTP.

- **Timeouts *(tempo limite)***

Timeout é a política suportada pelo Kuma onde define o tempo limite das conexões de saída.

- **Rate limite *(taxa de limite)***

É a política que faz proveito da limitação da taxa local do envoy para permitir a limitação das solicitações de serviços por instâncias. Todas as solicitações baseadas em Http/Http2 são suportadas.

A limitação de taxa local do envoy é uma política que aplica um limite de taxa de *bucket* de *token*, que basicamente é um algoritmo utilizado para controlar a transmissão de pacotes de dados em uma rede, ou seja, o limite de solicitações de taxas composto por cada solicitação processada pelo único *token*.
A configuração do rate limite permite configurar quantas solicitações serão permitidas num período de tempo específico e como o serviço irá responder quando esse limite for atingido.

Um cenário que explique essa situação:
Rate limite é uma política aplicada por instância de serviço, isto é, a partir do momento em que o serviço *backend* tiver uma taxa de 3 instância limitadas a cada 100 solicitações por segundo, significa que o serviço vai gerar uma taxa limitada de 300 solicitações por segundo.

- **Virtual Outbound *(saída virtual)***

Virtual *outbound* é a política utilizada pelo Kuma, que permite personalizar os nomes dos host e das portas de comunicação com os proxies de planos de dados.

O Kuma é concedido do resolvedor de DNS para fornecer nomeação de serviços, um mapeamento de nomes de hosts para IP's virtuais (VIPS).
Os IP's privados são alocados pelo plano de controle que vai devastando constantemente serviços que estão disponíveis em todas as malhas de serviços, de modo que quando um serviço é removido, o seu IP também é liberado e o Kuma DNS não responde por ele como um registo. Uma vez que um novo IP virtual é alocado, ou liberado, o plano de controle configura o plano de dados com essa alteração. De modo que todas as pesquisas de nomes são tratadas localmente pelo proxy do plano de dados e não pelo plano de controle, isso porque assim permite um manuseamento mais vigoroso na resolução de nomes.

O Kuma DNS não é uma forma de obter a descoberta de serviço, isto é, ele não retorna o endereço de IP real das instâncias dos serviços. Em vez disso, é retornado um VIP (IP virtual) alocado ao serviço relevante á malha, criando uma visão integra de todos os serviços num local ou em várias zonas.

Alguns casos de usos para essa política:
- Conservar o nome dos hosts utilizados na migração entre as malhas de serviços.
- Exposição de várias entradas em portas diferentes.

Algumas Limitações da saída virtual:
- Em saídas virtuais complexas o tráfego entre zonas não funciona, isso porque apenas as tags de serviços são distribuídas entre zonas.

- Em saídas virtuais, as combinações duplicadas de *hostname* e *port* que foram detetadas com prioridade mais alta é assumido.

O Kuma apresenta um conjunto de políticas, recursos de Monitoramento, onde englobamétricas, registos e rastreamento.

- Traffics Metrics (*Metricas de tráfego*)
- Service Map (*Mapa de serviço*)

O *traffic metrics* suportado pelo Kuma permite a facilidade de métricas de tráfego resistente em todos os proxys de plano de dados na sua malha de serviço, assim como também expor as métricas dos aplicativos executados ao lado dos proxies. Além de expor métricas dos proxies do plano de dado, também é capaz de expor métricas dos aplicativos executados ao lado dos proxie, ou seja, dos microserviços.

Através do *traffic mesh*, o Kuma tem suporte do painel de visualização que permite ver as estatísticas agregadas a uma única malha, assim como visualização de dependências de tráfego de serviço, que inclui informações como número de solicitações, taxas de erros.
O Kuma suporta integração com o Prometheus, ou seja, cada proxy pode expor as suas métricas em formato Prometheus. Assim o Prometheus também consegue automaticamente encontrar cada proxy na malha.

A coleta das métricas no Kuma, acontecem primeiro na exposição das métricas no proxies, e só depois configuração do Prometheus.
Prometheus, é uma plataforma de monitoramento de *open source*, que faz a coleção e armazenamento de métricas temporais. Ou seja, uma ferramenta de monitoramento de serviços e aplicações.

- **Traffic trace (*Rastreamento de tráfego*)**

A política de rastreamento permite fazer rastreamento dos *logs*. É uma política compatível para os seguintes protocolos: HTTP, HTTP2 e GRPC.
É possível especificar para cada proxy de serviço e plano de dados, qual o protocolo se quer habilitar o rastreamento.

- **Traffic Log (*Registro de tráfego*)**

O *traffic log* é a política que permite configurar e ter acesso aos *logs* de acesso em todos os planos de dados.

A essa configuração de *logs* de acesso compõe por três etapas:
1- *Adicicionar um back-end de registo*, é importante para a coleta de logs de acesso;
2- *Adicionar um recurso de trafficLog*, o recurso *trafficLog* é importante para o encaminhamento e acesso aos *logs*;
3- Agregação e visualização de logs;

Aos conjuntos de políticas e recursos avançados englobam configurações do envoy e segurança DP/CP

- Proxy template (*modelo de proxy*)
- DP/CP Security (*Segurança DP/CP*)

4.5 Gestão dos certificados pelo Kuma

A implementação do Kuma suporta mecanismos que fornecem um acesso seguro aos serviços através de:

- Segurança entre serviços através da política de segurança do TLS/mTLS.
- Segurança entre o controle plane do Kuma e os seus proxies de date plane, através do token do proxy do date plane.
- Proxy de *data plane* para controlar a comunicação do plano

Um proxy de *data plane* conecta-se ao *control plane* incluindo certificados mTLS.

O proxy da *data plane* e o *control plane* trocam informações confidenciais entre eles. Por padrão o servidor do *control plane* que é consumido pelo proxy da data *plane* é protegido por TLS com certificados criados automaticamente.

Segundo a documentação da NIST, a autenticação e a política de acesso dependem de qual API está sendo utilizado pelas microserviços, podem ser tanto, públicas, privadas, ou ate mesmo API de terceiros, que são API's exclusivo para parceiros de negócios. Aos vários microserviços, e às políticas de autenticação, devem ser definidas afim de cobrir todos eles.
A autenticação baseada em certificados, exige uma infraestrutura de chave publica (PKI) para o gerenciamento de certificados e chaves de gestão.

Entre os diversos tipos de certificados digitais, de uma forma breve e simples, descreve se como assinaturas digitais que são validadas por uma autoridade certificadora, que geram proteções para transações e serviços, o Kuma fornece os seguintes back-ends suportados pela CA (certificate authority)

- Uso de CA builtin;
- Uso de CA provided;

No caso do CA builtin, o certificado e a chave são gerados automaticamente, isto é, o Kuma gere automaticamente o seu próprio certificado raiz e chave que vai ser utilizador para provisionar ou alterar os certificados automaticamente para cada serviço, enquanto que no modo CA *provided*, o certificado e a chave são fornecidos pelo utilizador.

Ao utilizar um certificado e uma chave facultativa, para uma *back-end provid*, temos que garantir que cumpre alguns requisitos como:

- Tem de ser um certificado de CA raiz auto assinado, ou seja, certificados de CA intermediário não são permitidos), isto porque o uso de um certificado CA intermediário, fornece uma estrutura flexível de validade da confiança dos certificados de entidades intermediários, atuando como um *buffer* entre o certificado raiz e o servidor da entidade final, enquanto CA raiz é um certificado emitido diretamente por uma autoridade de certificação, ao contrário de outros certificados.

- Restrição básica do CA definida como *true*, essa restrição permite identificar de um certificado é um certificado de assinatura, assim como certificar a profundidade máxima de caminhos de certificados válidos incluem nesse certificado [57].

Um novo certificado de proxy de plano de dados é gerado automaticamente quando:
- Um proxy de plano de dados for reiniciado;
- Quando um plano de controle foi reiniciado;

Quando o proxy de plano de dados se conecta com um novo plano de controle;

4.6 Sumário

Neste capítulo, foi abordado uma análise da framework do Kuma, que entre as diferentes abordagens do *service mesh*, foi escolhida para análise, implementação, configurações e testes dos microserviços.
Foi feito uma análise da framework do Kuma, concretamente, o seu funcionamento, as políticas que por ele são suportadas, o que contribui para posteriormente avaliar a performance de microserviços e quão eficientes são os mecanismos de segurança do Kuma para aplicações inseguras.

Capítulo 5

Experimentação simples da aplicação de demonstração do Kuma

Neste capítulo inclui a análise do estado da arte relativamente à modelação dos serviços do redis e da aplicação de demonstração - demo app, em uma experimentação no Kuma.

5.1 Contexto

De acordo com a documentação do Kuma, ele pode ser executado em ambientes como: kubernetes, Docker, Centos, macOS, Ubuntu, entre outros [4].
O Kuma, sendo uma plataforma de código aberto, que foi arquitetado para funcionar em ambientes kubernetes[2], e VMs, que faz o gerenciamento de malhas de serviços e microserviços com suporte em várias zonas em muitos clusters, data centers e nuvens diferentes.

Nesse plano de estudo, o Kuma vai ser executado em uma VM (máquina virtual) com o sistema operativo Ubuntu, sendo analisado através da exploração e configuração de uma demo-app demonstração universal.
Este experimento tem como objetivo familiarizar, assim como aprender como configurar e aplicar os conceitos no framework do Kuma.

Na figura 1, temos o esquema da exploração do Kuma num aplicativo de demonstração universal, que consiste em dois serviços.

A app demonstração consiste em doi serviços { -Demo app; -Redis }

- Demo app (app web que permite incrementar um contador numérico)
- Redis (armazenamento de dados para o contador)

[2] https://www.vmware.com/topics/glossary/content/kubernetes.html

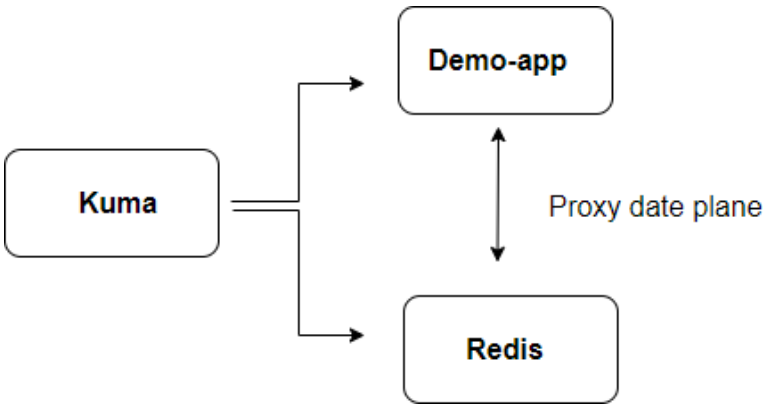

Figura 3: Exploração do Kuma com o aplicativo de demonstração universal

O redis oferece um conjunto de estruturas versáteis de dados na memória que permite a facilidade de criação de várias aplicações personalizadas.

Redis-Base de dados relacional focado em alto desempenho, ágil com acesso a armazenamento de informações. Altamente recomendado por aplicações que exigem um processamento dinâmico.
O Proxy é o que garante que o tráfego seja encaminhado para o serviço correto de destino.
E **o proxy date plane,** é projetada para conectar a um APi que faz a distribuição das configurações entre os proxies que estão no plano de dados.

O Kuma fornece capacidade de serviços tanto multi-zone, e localmente, nessa fase experimental, vai ser explorado numa zona local.

*No redis, a configuração é feita com um *daemon* na porta 26379 e definir o nome da zona.
Kuma (Kuma -cp) -controle plane
Kuma (Kuma -dp) -date plane

5.2 Configurações da exploração do demo-app no framework Kuma

A seguir descreve um conjunto de passos que foram executados para a instalação e configuração do Kuma.

1- Configuração e exploração do aplicativo de demonstração universal no ubuntu
Pré-requisitos:

- Kuma instalado;
- Redis e Demo-app instalado [50]

Demo-app: é a aplicação web que vai dar permissão da incrementação do contador numérico.
Redis: é o armazenamento de dados. Um armazenamento que fornece suporte de estrutura de dados como, *strings*, *hashes*, listas, *Scripts*, entre outros.

1.1 - Instalação do Kuma numa máquina virtual [49].

Antes da instalação do Kuma, foi preparada uma máquina virtual com o sistema operativo ubuntu 20.04.

Para a execução do Kuma foram efetuados esses três passos a seguir:

1. Download do Kuma: *curl -L https://Kuma.io/installer.sh | sh -*
2. Extrair o arquivo: *tar xvzf Kuma-*.tar.gz*
3. Executar o Kuma: *cd Kuma-1.3.1/bin*

Na fig. 4 podemos ver o conteúdo executável do Kuma pronto para ser utilizado.

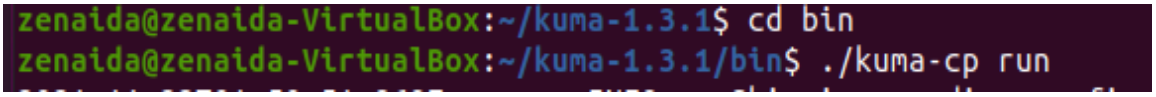

Figura 4: Conteúdo executável do Kuma

Apos executar o Kuma, podemos aceder ao plano de controle por meio do browser através de: ***127.0.0.1:5681/gui,*** como é ilustrado na fig. 5.

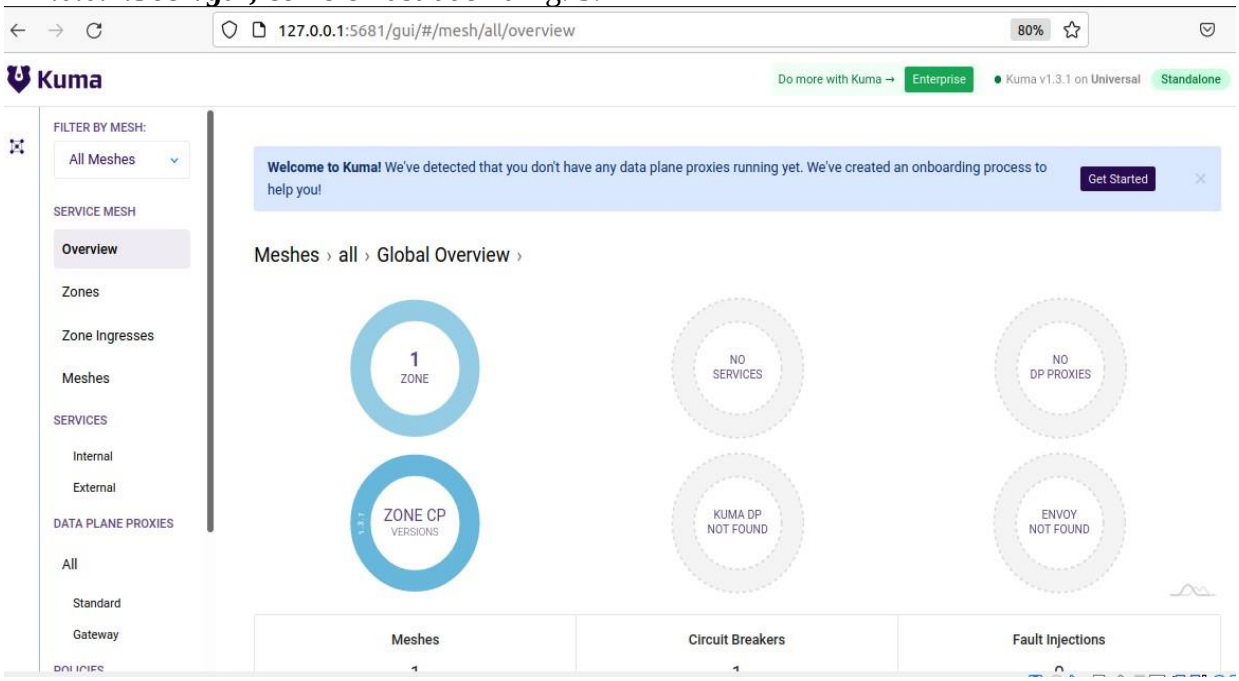

Figura 5: Interface do Kuma

Nesta primeira etapa, apresenta-se a página inicial, na fig. 6 onde irá iniciar a criação do nosso serviço.
Para essa fase experimental, estamos a utilizar uma malha de serviço por defeito.

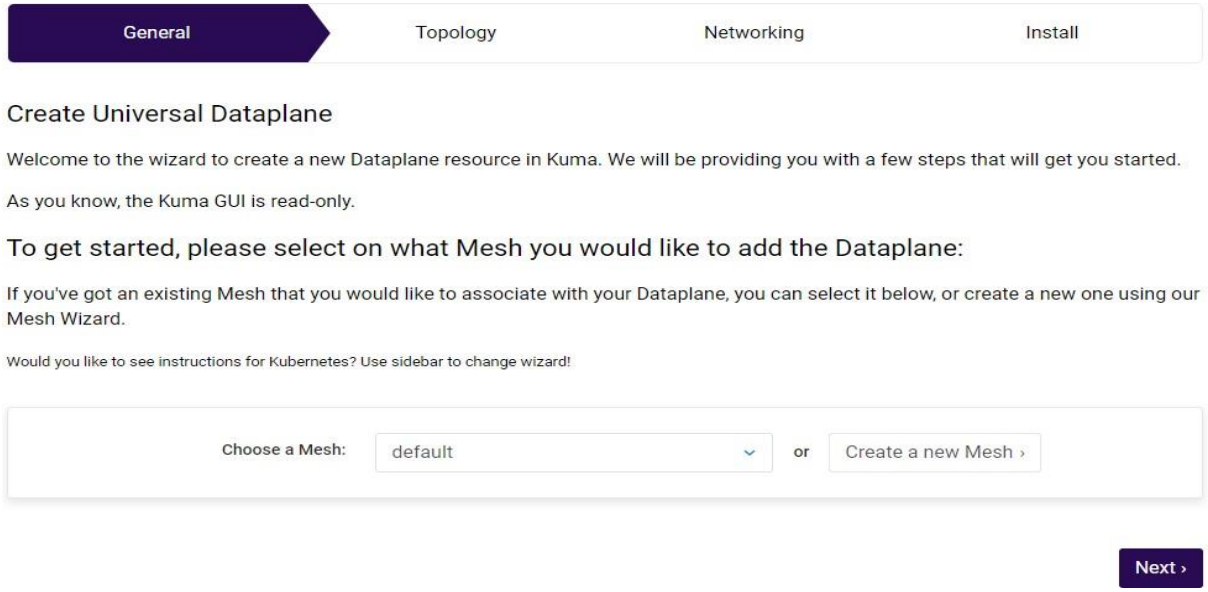

Figura 6: Interface de criação de serviço do Kuma

A seguir na Fig.6 demonstra o *setup dataplane* em uma topologia do Kuma.
Os *service mesh* são compostas por dois componentes: um **plano de controle,** e um **plano de dados**, que é responsável pelo fluxo de dados do sistema, de modo que cada consulta seja processada pelo plano de dados.
Enquanto que o plano de controle é responsável por coletar e armazenar e supervisiona os proxies no plano de dados, fornecendo uma API que os interliga.

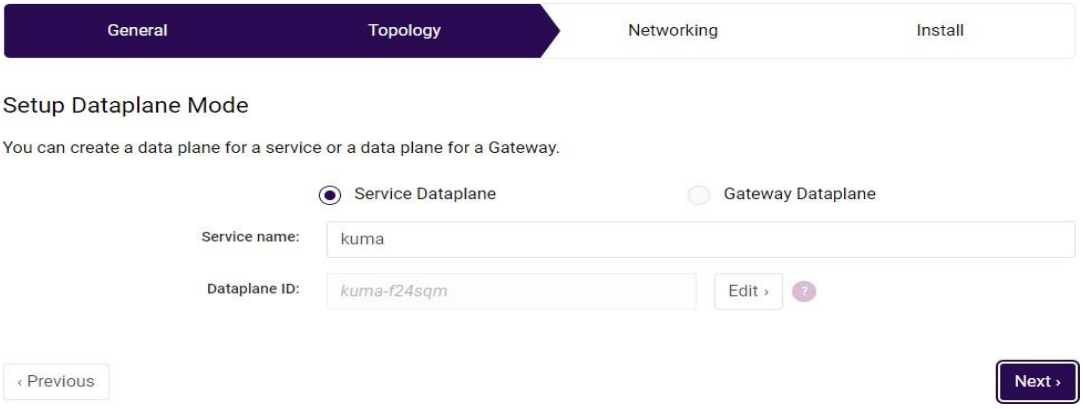

Figura 7: Setup dataplane mode do Kuma

Relembrando que estamos utilizando uma *service mesh* por defeito, isto é, um plano de controle autônomo, ou seja, um modo padrão de uma zona única, a fig. 8 mostra as configurações a serem feitas.

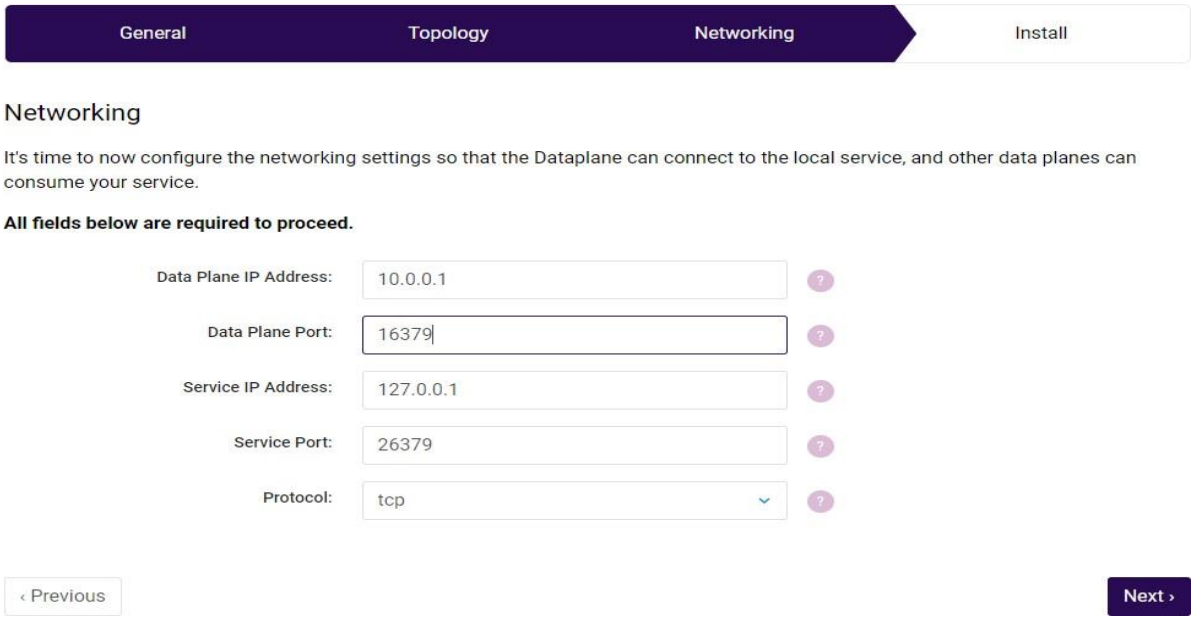

Figura 8: Configuração do Networking

Como pode-se ver na fig. 9, o auto-inject DPP, basicamente é o gerenciamento automático das credenciais que o Kuma irá utilizar para que seja permitido a autenticação entre o plano de dados e o plano de controle.

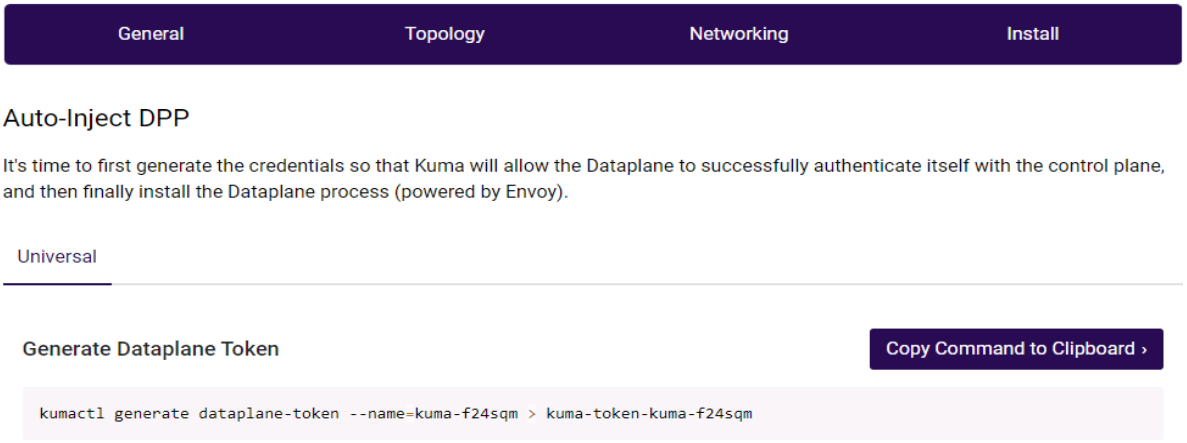

Figura 9: Criação dos tokens

2 - Instalação e configuração do *redis*.
2.1- Para a Instalação do redis é executada os seguintes códigos.
$ sudo add-apt-repository ppa:redislabs/redis
$ sudo apt-get update
$ sudo apt-get install redis

Na fig. 10, temos o output da instalação do redis.

```
zenaida@zenaida-VirtualBox:~/kuma-counter-demo$ sudo add-apt-repository ppa:redislabs/redis
 Redis is an open source (BSD licensed), in-memory data structure store, used as a database, cache and message broker.

It supports data structures such as strings, hashes, lists, sets, sorted sets with range queries, bitmaps, hyperloglogs, geospatial
indexes with radius queries and streams.

Redis has built-in replication, Lua scripting, LRU eviction, transactions and different levels of on-disk persistence, and provides
high availability via Redis Sentinel and automatic partitioning with Redis Cluster.
 Mais informação: https://launchpad.net/~redislabs/+archive/ubuntu/redis
Prima [ENTER] para continuar ou Ctrl+C para cancelar a sua adição.

Atg:1 http://pt.archive.ubuntu.com/ubuntu focal InRelease
Atg:2 http://pt.archive.ubuntu.com/ubuntu focal-updates InRelease
Atg:3 http://pt.archive.ubuntu.com/ubuntu focal-backports InRelease
Obter:4 http://ppa.launchpad.net/redislabs/redis/ubuntu focal InRelease [18,0 kB]
Obter:5 http://security.ubuntu.com/ubuntu focal-security InRelease [114 kB]
Obter:6 http://ppa.launchpad.net/redislabs/redis/ubuntu focal/main i386 Packages [492 B]
Obter:7 http://ppa.launchpad.net/redislabs/redis/ubuntu focal/main amd64 Packages [1016 B]
Obter:8 http://ppa.launchpad.net/redislabs/redis/ubuntu focal/main Translation-en [584 B]
Obtidos 134 kB em 2s (57,9 kB/s)
A ler as listas de pacotes... Pronto
zenaida@zenaida-VirtualBox:~/kuma-counter-demo$ sudo apt-get update
Atg:1 http://pt.archive.ubuntu.com/ubuntu focal InRelease
Atg:2 http://ppa.launchpad.net/redislabs/redis/ubuntu focal InRelease
Atg:3 http://pt.archive.ubuntu.com/ubuntu focal-updates InRelease
Atg:4 http://pt.archive.ubuntu.com/ubuntu focal-backports InRelease
Obter:5 http://security.ubuntu.com/ubuntu focal-security InRelease [114 kB]
Obtidos 114 kB em 6s (19,6 kB/s)
A ler as listas de pacotes... Pronto
zenaida@zenaida-VirtualBox:~/kuma-counter-demo$ sudo apt-get install redis
A ler as listas de pacotes... Pronto
```

Figura 10: Instalação do redis

2.2 -A seguir o próximo passo é execução do redis como um *daemon* na porta 26379 e definição de zona padrão
$redis-server --port 26379 --daemonize yes
$redis-cli -p 26379 set zone local

3 - Configuração do aplicativo de demonstração universal do Kuma
3.1-A instalação desta aplicação tem o propósito de verificar o funcionamento do Kuma. Primeiramente é preciso fazer o clone do Kuma-counter-demo: *git clone* https://github.com/Kumahq/Kuma-counter-demo.git, como é ilustrado na fig 11.

```
zenaida@zenaida-VirtualBox:~$ git clone https://github.com/kumahq/kuma-counter-demo.git
Cloning into 'kuma-counter-demo'...
remote: Enumerating objects: 124, done.
remote: Counting objects: 100% (124/124), done.
remote: Compressing objects: 100% (88/88), done.
remote: Total 124 (delta 63), reused 84 (delta 30), pack-reused 0
Receiving objects: 100% (124/124), 87.07 KiB | 775.00 KiB/s, done.
Resolving deltas: 100% (63/63), done.
```

Figura 11: Clone do Kuma-counter-demo

3.2- Após que a instalação é feita a inicialização do demo-app na porta padrão 5000, com os seguintes códigos:
$npm install --prefix=app/
$npm start --prefix=app/

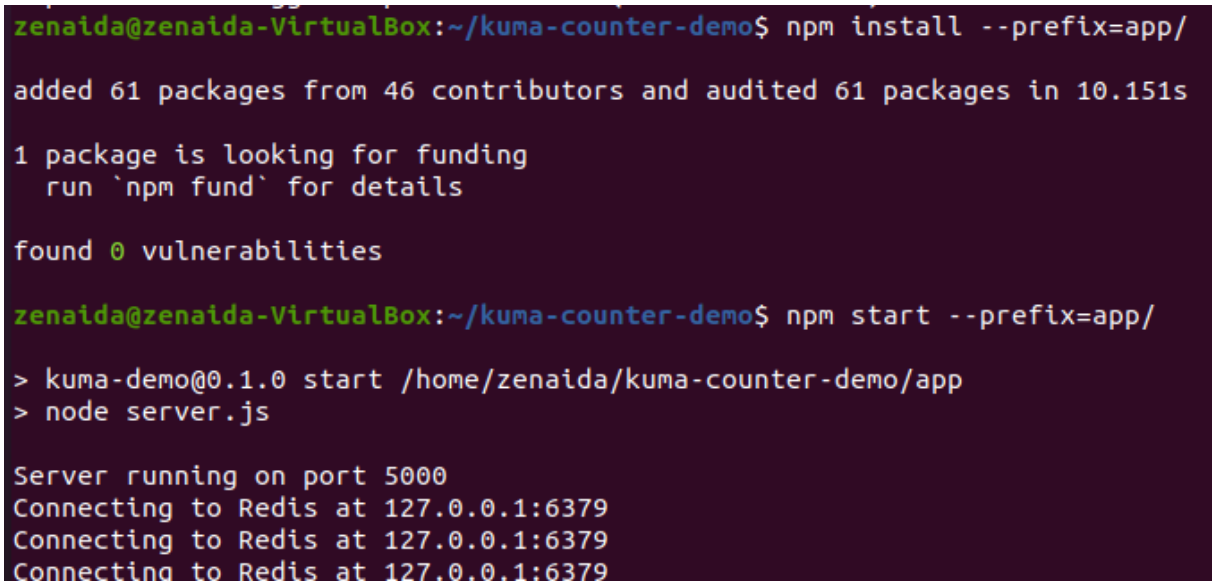

Figura 12: Inicialização do demo-app

A aplicação de demonstração é baseada na tecnologia *NodeJS*, como se pode ver na fig. 12.
A tecnologia *NodeJS*, é utilizada principalmente em servidores, e orientado a eventos, executando em um único processo sem bloqueio, permitindo que milhares de conexões simultâneas em um único servidor se conecte sem a necessidade de gerenciar simultaneamente *threads*, e desperdiçar ciclos d CPU [51] , ou seja o *NodeJS*, é uma tecnologia assíncrona que trabalha em uma única *thread* de execução, que não bloqueia a *thread* da execução, dando suporte assim a um grande volume de requisições ao mesmo tempo.

4 - Criação de proxy de plano de dados para cada serviço.
A criação do proxy nesta etapa serve para garantir o encaminhamento da mensagem entre o redis e o Demo-app.

Para Redis:

```
Kuma-dp run \
  --cp-address=https://localhost:5678/ \
  --dns-enabled=false \
  --dataplane-token-file=Kuma-token-redis \
  --dataplane="
  type: Dataplane
  mesh: default
  name: redis
  networking:
    address: 0.0.0.0
    inbound:
      - port: 16379
      servicePort: 26379
      serviceAddress: 127.0.0.1
      tags:
        Kuma.io/service: redis
        Kuma.io/protocol: tcp"
```

Para Demo:

```
Kuma-dp run \
  --cp-address=https://localhost:5678/ \
  --dns-enabled=false \
  --dataplane-token-file=Kuma-token-app \
  --dataplane="
  type: Dataplane
  mesh: default
  name: app
  networking:
    address: 0.0.0.0
    outbound:
      - port: 6379
        tags:
          Kuma.io/service: redis
    inbound:
      - port: 15000
        servicePort: 5000
        serviceAddress: 127.0.0.1
        tags:
          Kuma.io/service: app
          Kuma.io/protocol: http"
```

5.2 Sumário

Neste capítulo foi feito uma análise de uma experimentação simples da demo app na framework do Kuma. A análise feita anteriormente facilitou a experimentação documentada neste capítulo, permitindo perceber as configurações necessárias e mais pertinentes.

Agora através de uma experimentação simples, foi feito uma breve análise de como funciona. Foi utilizado dois serviços nessa demonstração, o demo app e o Redis. Em que o demo app é uma aplicação web, que vai comunicar com o Redis, que é um armazenamento de dados.
Através desses dois microserviços, foi feito uma breve análise, do funcionamento e configuração de algumas implementações como, por exemplo análise e criação de uma *mesh*, conexão dos microserviços através do *controlplane* e do *dataplane*.

Este experimento foi muito importante, pois permitiu a analise e o conhecimento do campo de trabalho. Apos esse experimento simples, para o próximo capítulo será a abordado de uma forma mais profundo a implementação de serviços no Kuma, incluindo configurações de um conjunto de recomendações face às diretrizes da NIST de forma a tornar um serviço seguro e resiliente.

Capítulo 6

Implementação de serviços do Kuma

Neste capítulo, será apresentado a parte prática, que contém a implementação e configuração de um conjunto de serviços com complexidade variável no framework do Kuma.
Este desenvolvimento consiste em implementações e configurações, apresentando e analisando resultados obtidos face às diretrizes de segurança da informação da NIST, com finalidade de fornecer segurança para infraestrutura, com desenvolvimento e diretrizes de gestão administrativo, técnicos para a segurança e a privacidade das informações.

6.1 Contexto

Neste contexto, vai ser realizado em prática um experimento de uma solução simples no Kuma, composto por dois front-end (Java), e a base de dados PostgreSQL (backend).
A malha de serviço é uma camada de infraestrutura fornecida para gerenciamento de comunicações entre serviços.

Neste experimento será utilizado**:**

- **PostgreSQL**: Uma base de dados, pronto para ser executado em Docker container comauxílio da ferramenta *docker-compose;*

- **Java**: Código fonte do template da aplicação web em java com o Docker container configurado, com uma api web, composta por um conjunto de funções definido de mensagens de requisitos e respostas *HTTP*, permitindo aceder a dados, pronto para ser executado em *docker-compose* com o PostgreSQL.

O serviço implementado consta com dois serviços, como descrito acima, uma implementação totalmente insegura, e vulnerável (sem suporte para HTTPS). O objetivo é, junto com os protocolos de segurança suportado pelo Kuma e as orientações da NIST, implementar configurações que permitam tornar a infraestrutura mais segura e resiliente.

6.2 Objetivos

Os objetivos deste componente prático são os seguintes:

- Avaliar o impacto na performance das aplicações (comparação do funcionamento dos microserviços no *service mesh*);
- Avaliar o impacto da monitorização do funcionamento dos microserviços;
- Avaliar o impacto das configurações de segurança mTLS e TLS;
- Avaliar a performance dos microserviços com diferentes políticas de tráfego;
- Avaliação do impacto da resiliência e performance do microserviços na presença de falhas.

6.3 Implementação dos serviços em Kuma

Na fig.13 apresenta o esquema da implementação dos serviços no Kuma, primeiramente é preciso criar as imagens dos microserviços, e posteriormente a criação dos *templates yaml* para as imagens criadas.

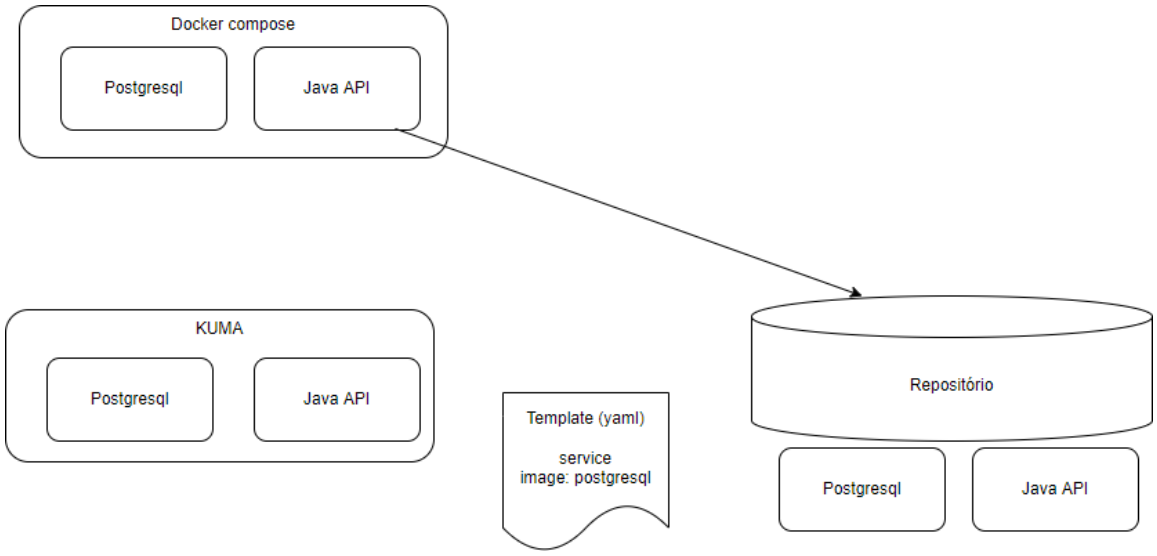

Figura 13: Implementação do serviço no Kuma

O objetivo desta demonstração é dar a conhecer os microserviços e perceber como eles interagem entre eles através de uma API.
A demonstração do API REST java demonstração que está a ser utilizado, utiliza o *Spring boot*[3] que é uma das soluções utilizadas para desenvolver API's REST e microserviços. O *Spring boot* facilita a criação de aplicativos independentes baseados em Spring de nível de produção de execução.

Para a demonstração deste experimento, vamos dividir em três partes:

1. A primeira parte consiste em executar e analisar os serviços no Docker;
2. A segunda parte consiste em criação do repositório das imagens dos serviços;
3. A terceira parte será a contextualização de colocar os serviços a funcionar juntamente com o Kuma.

6.4 Configurações dos serviços no Kuma

1-Execução dos serviços *PostgreSQL* e *Java* no Docker

Nesta fase, de execução e análise dos serviços no Docker, para executar este projeto são necessários alguns requisitos. Requisitos para execução dos serviços no Docker:

- Docker
- Docker-compose

1.1-Para iniciar essa demonstração, deve executar o script:
./docker-compose-java-psql.sh, para ter o servidor e a base dados em execução.

```
PS D:\Documentos\services-main\api-demo> ./docker-compose-java-psql.sh
PS D:\Documentos\services-main\api-demo>
```

Figura 14: Execução do script do java

[3] https://spring.io/projects/spring-boot

1.2- A figura seguinte nos dá o acesso ao Web browser através de: http://localhost:8080

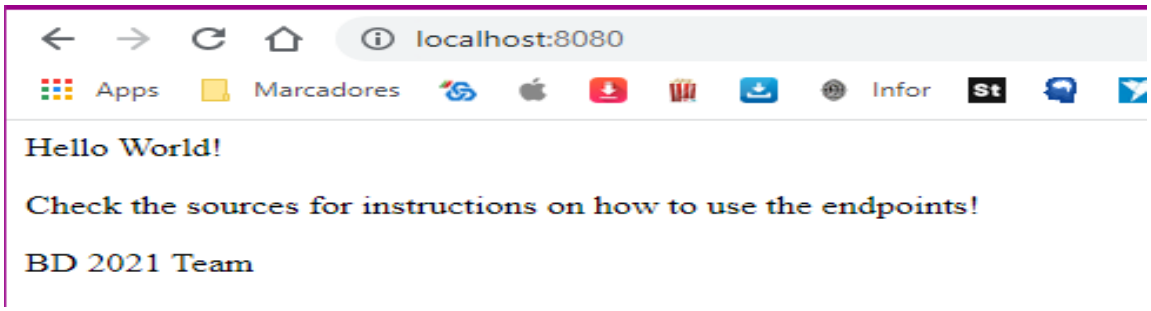

Figura 15: Acesso web browser

1.3- Para correr o serviço, basta correr o script: *docker-compose -f docker-compose-java-psql.yml up --build*, como aparece na fig.16 a seguir.

```
PS D:\Documentos\services-main\api-demo> docker-compose -f docker-compose-java-psql.yml up --build
[+] Building 439.7s (17/17) FINISHED
 => [api-demo_db internal] load build definition from Dockerfile                                                                0.4s
 => => transferring dockerfile: 375B                                                                                            0.1s
 => [api-demo_web internal] load build definition from Dockerfile                                                               0.4s
 => => transferring dockerfile: 403B                                                                                            0.1s
 => [api-demo_db internal] load .dockerignore                                                                                   0.4s
 => => transferring context: 2B                                                                                                 0.0s
 => [api-demo_web internal] load .dockerignore                                                                                  0.3s
 => => transferring context: 2B                                                                                                 0.0s
 => [api-demo_db internal] load metadata for docker.io/library/postgres:latest                                                  5.4s
 => [api-demo_web internal] load metadata for docker.io/library/maven:3-jdk-11-slim                                             4.9s
 => [auth] library/postgres:pull token for registry-1.docker.io                                                                 0.0s
 => [auth] library/maven:pull token for registry-1.docker.io                                                                    0.0s
 => [api-demo_web internal] load build context                                                                                  0.2s
 => => transferring context: 16.90kB                                                                                            0.1s
 => [api-demo_web 1/4] FROM docker.io/library/maven:3-jdk-11-slim@sha256:04f8e5ba4a6a74fb7f97940bc75ac7340520728d2fb051ecc5c9ecbb9ba2  342.7s
 => => resolve docker.io/library/maven:3-jdk-11-slim@sha256:04f8e5ba4a6a74fb7f97940bc75ac7340520728d2fb051ecc5c9ecbb9ba28b48    0.1s
 => => sha256:04f8e5ba4a6a74fb7f97940bc75ac7340520728d2fb051ecc5c9ecbb9ba28b48 549B / 549B                                      0.0s
 => => sha256:4c5742fa41cdeea437e22332db82f5e397acf400374d339f8166053badb80f04 2.00kB / 2.00kB                                  0.0s
 => => sha256:da9a1aaba88d4795a4b9ae09ba7d3d3177fe12a5363ade07b281cc67da9a7a16 8.51kB / 8.51kB                                  0.0s
 => => sha256:ea8366d5a4a5f71bf009ed651eb5566bf81bcd61ddaa4a2541fc65505bb076ca 1.58MB / 1.58MB                                  2.3s
 => => sha256:f7a1c6dad28192bd417b78079d6ddc03cbca6d5ea46bba12769b235b6353c00c 31.37MB / 31.37MB                               96.4s
 => => sha256:e3634e671f7dc632d921a097fe73b18827d8bb820de8629789b27342956ea45e 211B / 211B                                      1.9s
 => => sha256:3c55cecd453e8a40e0be1c4897a85a28e9a7a2cbde86f24352e1530fc01ed252 203.53MB / 203.53MB                            332.2s
 => => sha256:23316b953783ce83ffa6bb2cfa55b19a6d1ed9d99ea3bbb3fbfb678659ea55bd 2.46MB / 2.46MB                                  6.5s
```

Figura 16: Execução do script

2- Criação das imagens dos serviços *PostgreSQL e Java* no repositório do Docker

Nesta parte consiste na criação da imagem *PostgreSQL* no Docker, precisamos ter a certeza de que estamos no diretório ***PostgreSQL.***

2.1- Criação da imagem no Docker – *PostgreSQL*

O processo da criação do *PostgreSQL image* vai ser exibida nos seguintes screenshots.

Na figura fig.17 temos o screenshot da criação da imagem *PostgreSQL*, com o comando *docker build . -t zenaida94/PostgreSQL*

```
PS D:\Documentos\services-main2\api-demo\postgresql> docker build . -t zenaida94/postgresql
[+] Building 2.7s (8/8) FINISHED
 => [internal] load build definition from Dockerfile
 => => transferring dockerfile: 375B
 => [internal] load .dockerignore
 => => transferring context: 2B
 => [internal] load metadata for docker.io/library/postgres:latest
 => [auth] library/postgres:pull token for registry-1.docker.io
 => [internal] load build context
 => => transferring context: 37B
 => [1/2] FROM docker.io/library/postgres@sha256:ab0be6280ada8549f45e6662ab4f00b7f601886fcd55c5976565d4636d87c8b2
 => CACHED [2/2] COPY BD2021_data.sql /docker-entrypoint-initdb.d/
 => exporting to image
 => => exporting layers
 => => writing image sha256:b792f41b83a62d396a69e15f82bbb08eaa835addf18d12f3c1672a21ae486c44
 => => naming to docker.io/zenaida94/postgresql
```

Figura 17: Docker build PostgreSQL-image

Na fig.18 apresenta o *Docker push*, um comando utilizado para enviar ou partilhar uma imagem no Docker ou de um repositório através de um registo que pode ser publico ou privado.

```
Use 'docker scan' to run Snyk tests against images to find vulnerabilities and learn how to fix them
PS D:\Documentos\services-main2\api-demo\postgresql> docker push zenaida94/postgresql
Using default tag: latest
The push refers to repository [docker.io/zenaida94/postgresql]
512bc944303b: Pushed
a17023a68ac1: Pushed
f689b0784ee9: Pushed
65a24bc36638: Pushed
bb50f3907a94: Pushed
1c17577cdf5c: Pushed
12e6d5c212c5: Pushed
54ccbeacfc8c: Pushed
eb4827143dd5: Pushed
9829cf46cc7b: Pushed
289452c265f5: Pushed
8f6516bbd7c3: Pushed
f121e8841357: Pushed
9c1b6dd6c1e6: Pushed
latest: digest: sha256:73fa3769d40d0db9674568b87c301fe2e33d49f59d2e5e39d69e53317f2344a8 size: 3247
PS D:\Documentos\services-main2\api-demo\postgresql>
```

Figura 18: Docker push do PostgreSQL

2.2- Criação da imagem no Docker – Java e java1

A criação da imagem *Java* é o mesmo procedimento do *PostgreSQL*.

O processo da criação do *Java image* vai ser exibida nos seguintes screenshots, [fig.19 – fig. 20], documentando os seguintes passos:

Passo 1- Docker *build*, que é a construção do Java *Image*;

Passo 2- Docker *push*, é enviar a imagem para o repositório do Docker.

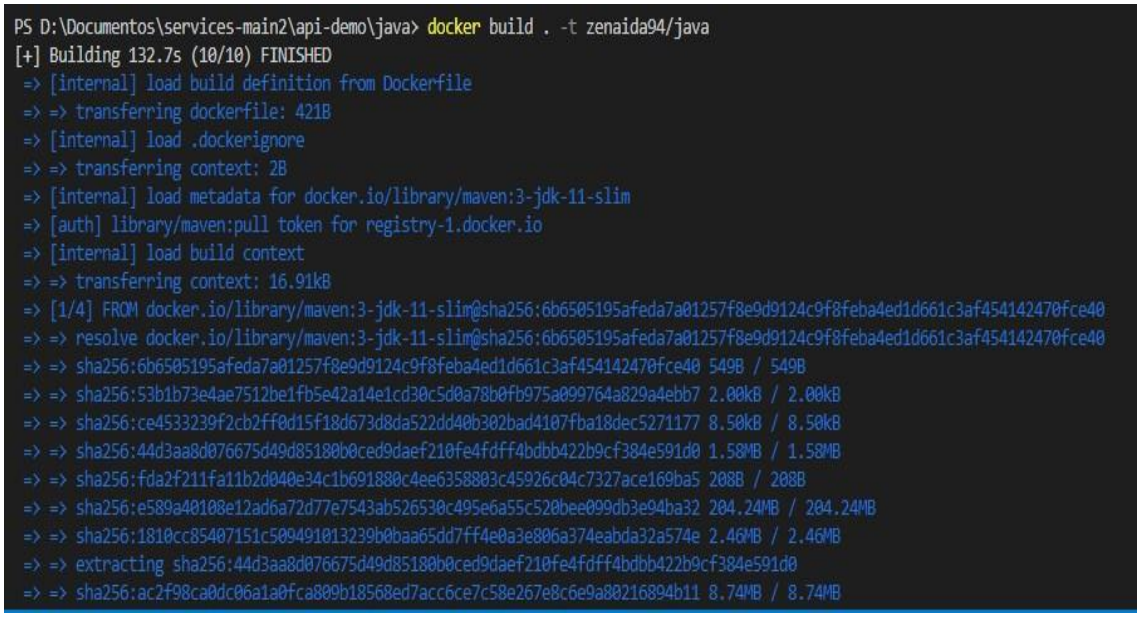

Figura 19: Docker build java

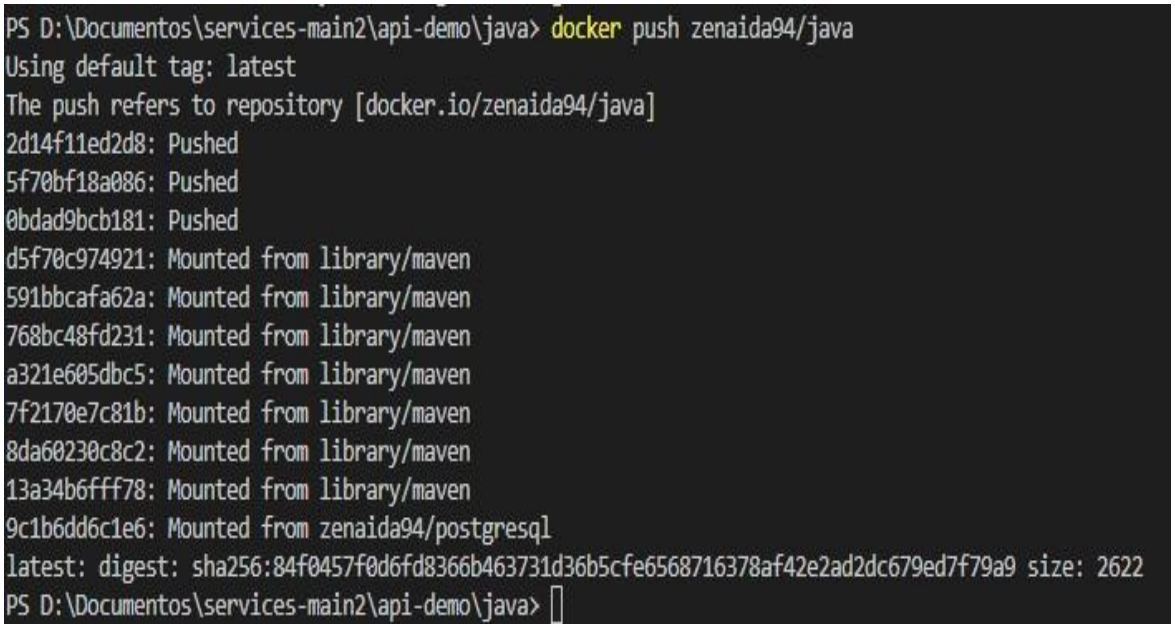

Figura 20: Docker push do java

Na fig. 21, mostra as configurações dos serviços que vão ser utilizados, através do Docker compose.

```
 4 #
 5 # Authors:
 6 #   Nuno Antunes <nmsa@dei.uc.pt>
 7 #   BD 2021 Team - https://dei.uc.pt/lei/
 8 #   University of Coimbra
 9 version: '3'
10
11 services:
12   db:
13     build: ./postgresql
14     container_name: db
15     expose:
16       - "5432"
17     ports:
18       - "5432:5432"
19   web:
20     build: ./java
21     container_name: api
22     expose:
23       - "8080"
24     ports:
25       - "8080:8080"
26     depends_on:
27       - db
28
29   web1:
30     build: ./java1
31     container_name: api1
32     expose:
33       - "34412"
34     ports:
35       - "34412:34412"
36     depends_on:
37       - db
```

Figura 21: Configuração do ficheiro Dockerfile

Com a Fig. 22, podemos observar a lista das imagens criadas.

```
zenaida@zenaida-VirtualBox:~/services/api-demo$ docker ps
CONTAINER ID   IMAGE          COMMAND                  CREATED          STATUS          PORTS
  NAMES
54a47a7d2568   api-demo_web1  "/usr/local/bin/mvn-…"   55 minutes ago   Up 55 minutes   0.0.0.0:34412->34412/tcp,
  api1
0b35859ff8df   api-demo_web   "/usr/local/bin/mvn-…"   2 months ago     Up 2 months     0.0.0.0:8080->8080/tcp, :
  api
8e6b69f5ddc7   api-demo_db    "docker-entrypoint.s…"   2 months ago     Up 2 months     0.0.0.0:5432->5432/tcp, :
  db
zenaida@zenaida-VirtualBox:~/services/api-demo$
```

Figura 22: Comando *docker ps* para listar as imagens criadas

As figuras seguintes [23-26] apresentam os passos efetuados para a criação do serviço java1:
Passo 1 – Criação do dataplane do microserviço java1, ilustrado na fig.23

Passo 2 – Configuração do dataplane anteriormente criado, ilustrado na fig.24
Passo 3 – Execução do Token para java1, ilustrado na fig.25.
Passo 4 – Setup do dataplane, ilustrado na fig.26

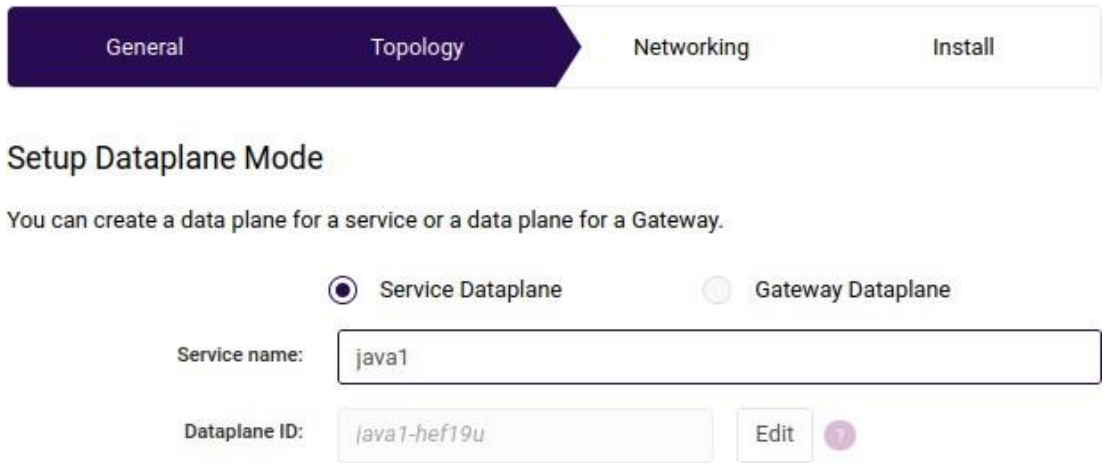

Figura 23: Criação do dataplane Java1

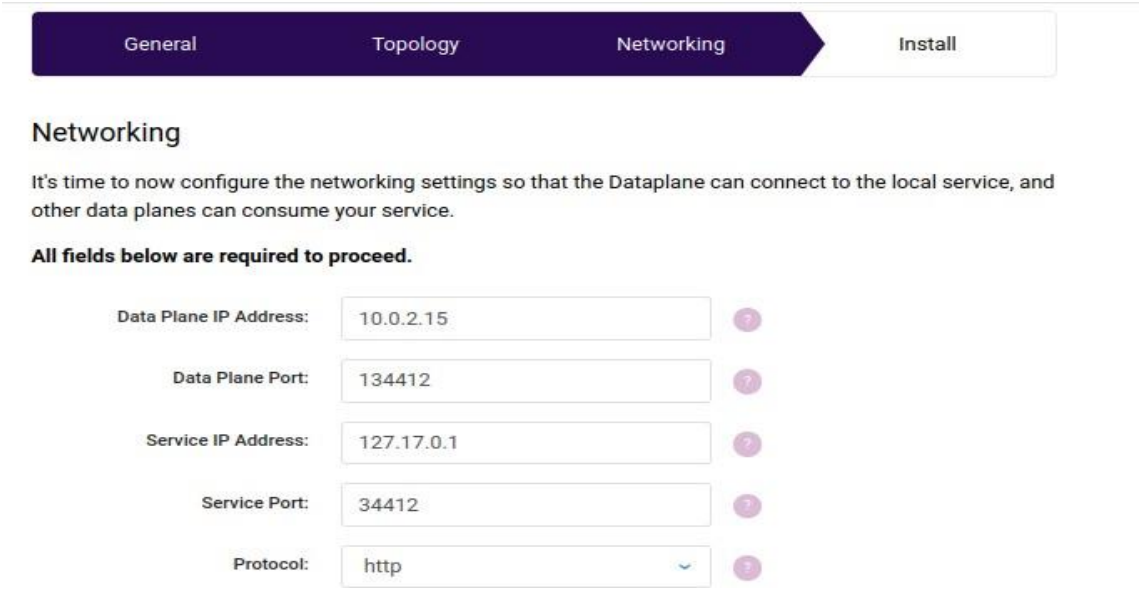

Figura 24: Setup do dataplane java1

```
zenaida@zenaida-VirtualBox:~$ cd kuma-1.3.1/
zenaida@zenaida-VirtualBox:~/kuma-1.3.1$ ./bin/kumactl generate dataplane-token
--name=java1-hef19u > kuma-token-java1-hef19u
```

Figura 25: Execução do token java1

```
zenaida@zenaida-VirtualBox:~/kuma-1.3.1$ ./bin/kuma-dp run       --cp-address=https://127.0.0.1:5678/
      --dataplane="type: Dataplane
mesh: default
name: java1-hef19u
networking:
  address: 10.0.2.15
  inbound:
    - port: 34412
      servicePort: 34412
      serviceAddress: 127.17.0.1
      tags:
        kuma.io/service: java1
        kuma.io/protocol: http"       --dataplane-token-file=kuma-token-java1-hef19u
```

Figura 26: Setup do run dataplane java1

3- Execução dos serviços *PostgreSQL* e *Java* no Kuma

Após a análise e execução dos serviços no docker e avaliação de como eles se comunicam, nessa parte vai ser apresentado um conjunto de passos com instruções de como colocar os serviços (PostgreSQL e Java) em funcionamento no Kuma.
O Kuma fica também configurado com persistência como foi visto no capítulo 4, guardando as configurações na base de dados PostgreSQL.

3.1- Ter os serviços a correr

Para este caso, primeiramente é correr com o docker-compose os serviços que estão no repositório do githubt.

- Instalar o Docker-compose;
- Fazer o git clone dos serviços https://github.com/nmsa/services.git;
- Setup and run dos serviços

  sh build.sh – to build the Docker image
  sh run.sh – to run the container

Nas figuras seguintes [27-30] apresenta os passos efetuados para fazer o setup e o run dos serviços.
Passo 1- Compilação do script PostgreSQL, como ilustrado na fig. 27.
Passo 2- Compilação do script do Java, como ilustrado na fig. 28.
Passo 3- Serviço PostgreSQL compilado, em execução, como ilustrado na fig.29.
Passo 4- Todos os serviços em execução no Docker, como ilustrado na fig.30.

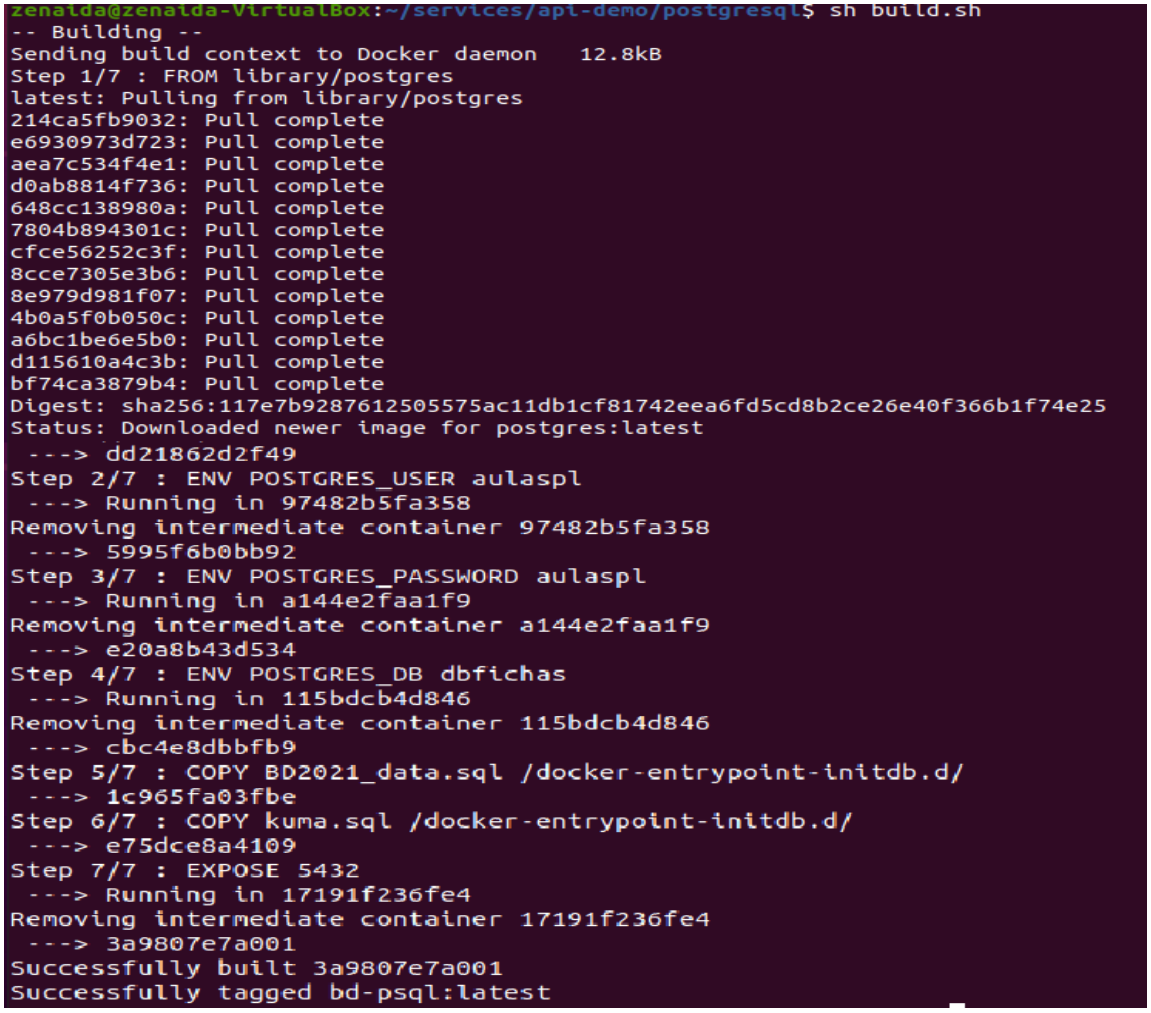

Figura 27: Sh build.sh PostgreSQL

Figura 28: Docker-composse-java-psql.sh

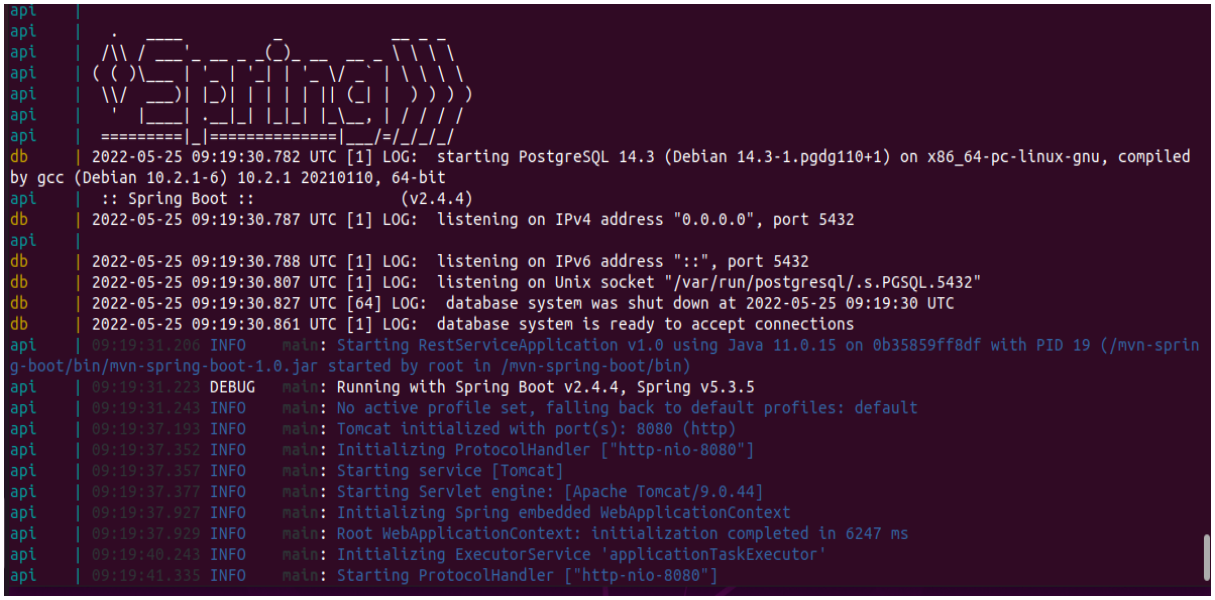

Figura 29: Service compiled and running

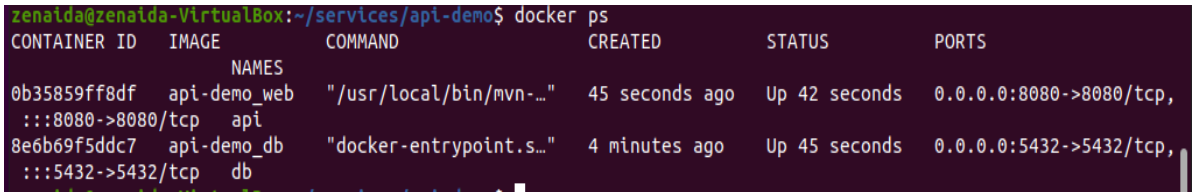

Figura 30: Os serviços a correr no Docker

3.2- Configuração da base dados do Kuma no PostgreSQL

Criação da estrutura da base de dados do Kuma

```
KUMA_STORE_TYPE=postgres \
  KUMA_STORE_POSTGRES_HOST=127.0.0.1 \
  KUMA_STORE_POSTGRES_PORT=5432 \
  KUMA_STORE_POSTGRES_USER=aulaspl \
  KUMA_STORE_POSTGRES_PASSWORD=aulaspl \
  KUMA_STORE_POSTGRES_DB_NAME=Kuma \
  ./Kuma-cp migrate up
```

Na fig. 31, podemos ver o resultado da criação e migração dos dados do Kuma no PostgreSQL.

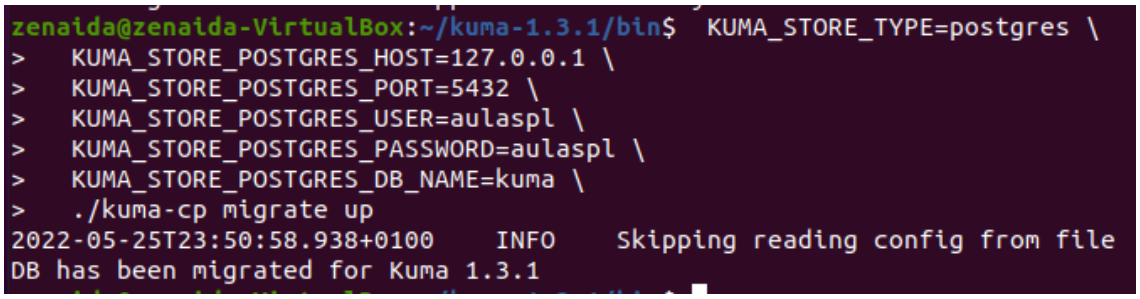

Figura 31: Migração dos dados do Kuma no PostgreSQL

3.3-Arrancar o Kuma com configuração persistente

Arranco com o plano de controlo do Kuma

```
KUMA_STORE_TYPE=postgres \
  KUMA_STORE_POSTGRES_HOST=127.0.0.1 \
  KUMA_STORE_POSTGRES_PORT=5432 \
  KUMA_STORE_POSTGRES_USER=aulaspl \
  KUMA_STORE_POSTGRES_PASSWORD=aulaspl \
  KUMA_STORE_POSTGRES_DB_NAME=Kuma \
  ./Kuma-cp run
```

Na fig. 32, podemos ver o resultado do arranco do Kuma ligado à base dados

```
zenaida@zenaida-VirtualBox:~/kuma-1.3.1/bin$  KUMA_STORE_TYPE=postgres \
>   KUMA_STORE_POSTGRES_HOST=127.0.0.1 \
>   KUMA_STORE_POSTGRES_PORT=5432 \
>   KUMA_STORE_POSTGRES_USER=aulaspl \
>   KUMA_STORE_POSTGRES_PASSWORD=aulaspl \
>   KUMA_STORE_POSTGRES_DB_NAME=kuma \
>   ./kuma-cp run
2022-05-26T00:04:12.910+0100    INFO    Skipping reading config from file
2022-05-26T00:04:12.918+0100    INFO    bootstrap.auto-configure        directory /home/zenaida/.kuma
will be used as a working directory, it could be changed using KUMA_GENERAL_WORK_DIR environment varia
ble
2022-05-26T00:04:12.920+0100    INFO    bootstrap.auto-configure        Kuma detected TLS cert and key
 in the working directory
```

Figura 32: Arranque do Kuma ligado à base de dados

3.4- Criação dos planos de dados no Kuma para cada um dos microserviços

Para o plano de dados têm de ser especificados algumas informações importantes, conforme sumarizado na tabela 5.

Tabela 5 – Informações correspondendo aos serviços e dataplane

Campo	Descrição
Data Plane IP Address	Corresponde ao IP local da máquina
O Data Plane Port	Corresponde ao IP que vai ser usado para aceder ao serviço através do Kuma
Service IP Address	Com o IP do Docker network, ondes está presentemente o PostgreSQL a correr
Service Port	Porto normal usado pelo PostgreSQL

3.4.1-Para app Java

Nas figuras seguintes apresentam um conjunto de passos, para a criação do serviço Java.

Passo 1- Apresentação do modo de criação do setup do dataPlane mode para o nosso microserviço java, ilustrado na fig. 33.
Passo 2- Configuração dos dados para o dataplane da app java, ilustrado na fig.34.
Passo 3- Gerenciamento do token que permitido a autenticação entre o plano de dados e o plano de controle, ilustrado na fig. 35.
Passo 4- Deploy da dataplane, ilustrado na fig.36.
Passo 5- Apresentação do resultado final, ilustrado na fig.37.

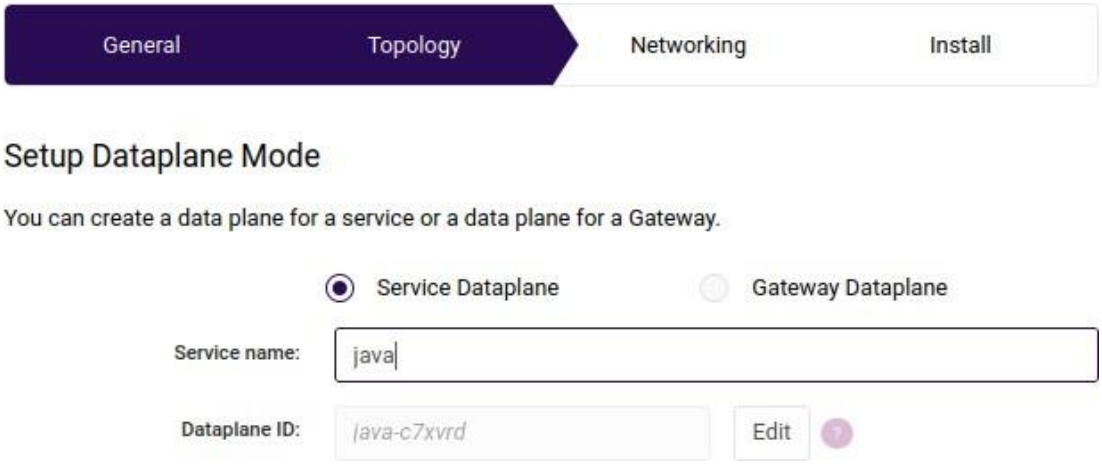

Figura 33: Setup do Dataplane Java

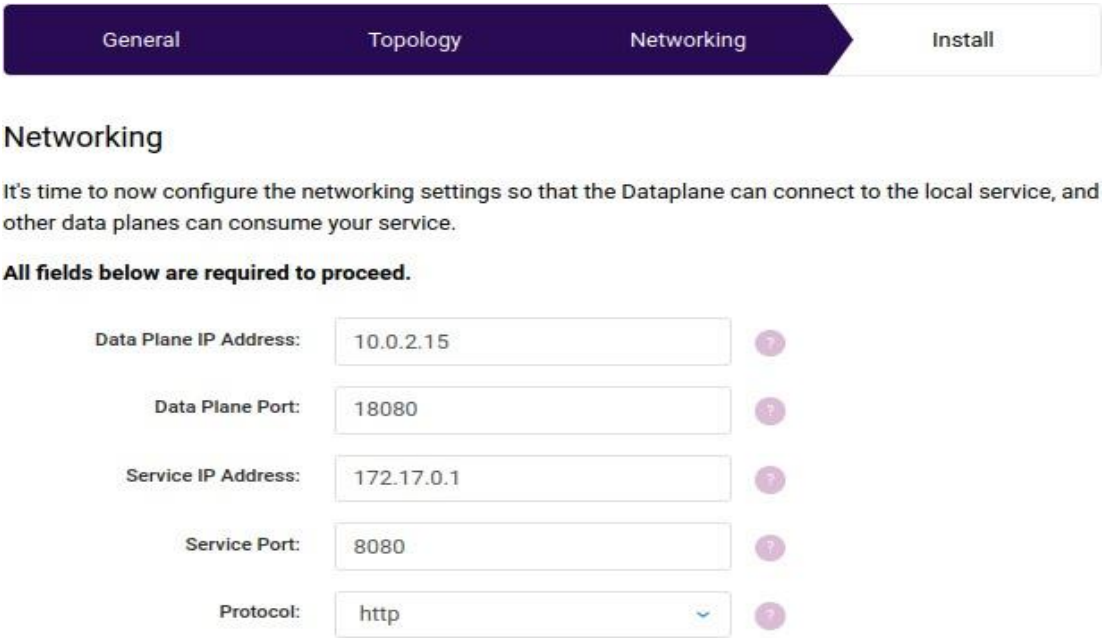

Figura 34: Configuração dos dados para a dataplane Java

Para finalizar, o Kuma faz o gerenciamento automático do *token*, que posteriormente será utilizado para que seja permitido a autenticação entre o plano de dados e o plano de controle, como é mostrado na fig. 35.

```
zenaida@zenaida-VirtualBox:~/kuma-1.3.1$ ./bin/kumactl generate dataplane-token --name=java-ohrji8
> kuma-token-java-ohrji8
zenaida@zenaida-VirtualBox:~/kuma-1.3.1$
```

Figura 35: Execução do token gerado para Java

Após a criação do token para a java app, o próximo a fazer é fazer o Kuma-run como é mostrado na fig. 36.

```
zenaida@zenaida-VirtualBox:~/kuma-1.3.1$ ./bin/kuma-dp run \
>       --cp-address=https://127.0.0.1:5678/ \
>       --dataplane="type: Dataplane
> mesh: default
> name: java-ohrji8
> networking:
>   address: 10.0.2.15
>   inbound:
>     - port: 18080
>       servicePort: 8080
>       serviceAddress: 172.17.0.1
>       tags:
>         kuma.io/service: java
>         kuma.io/protocol: http" \
>       --dataplane-token-file=kuma-token-java-ohrji8
2022-05-28T16:28:27.114+0100    INFO    Skipping reading config from file
```

Figura 36: Execução do comando para fazer deploy do dataplane

Após de seguir todos os passos, o resultado esperado é o seguinte, como mostra a fig. 37.

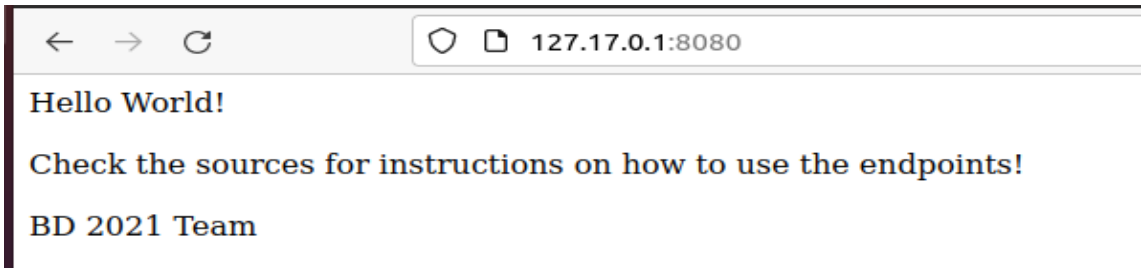

Figura 37: Resultado esperado

3.4.2- Para app PostgreSQL

Para a criação do serviço PostgreSQL, foi executado os mesmos passos utilizados para a criação do java, como ilustrado nas figuras anteriores.

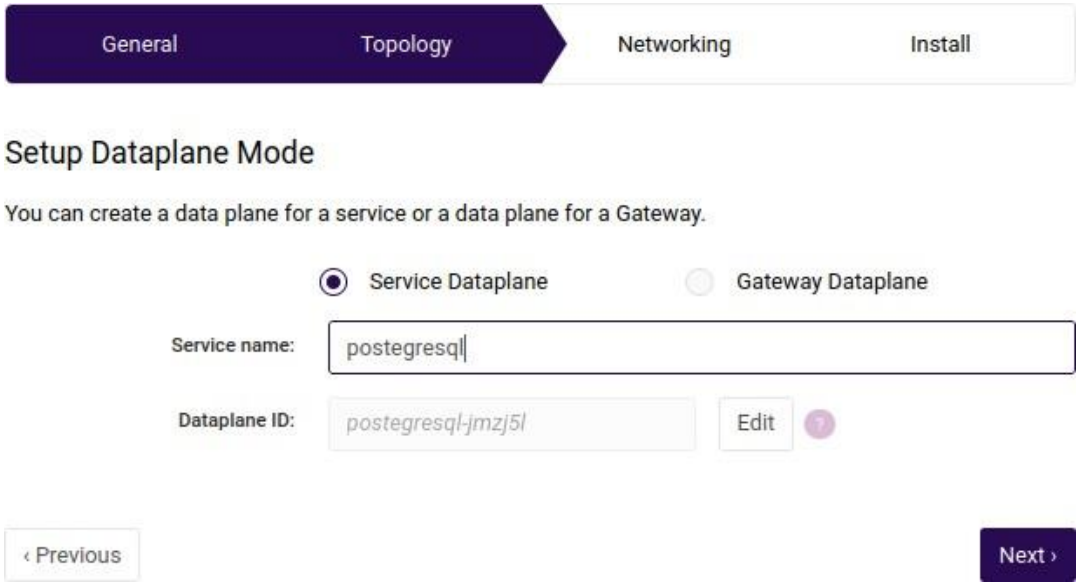

Figura 38: Setup Dataplane mode para o PostgreSQL service

Na fig.39 temos a configuração dos dados para o dataPlane da app PostgreSQL

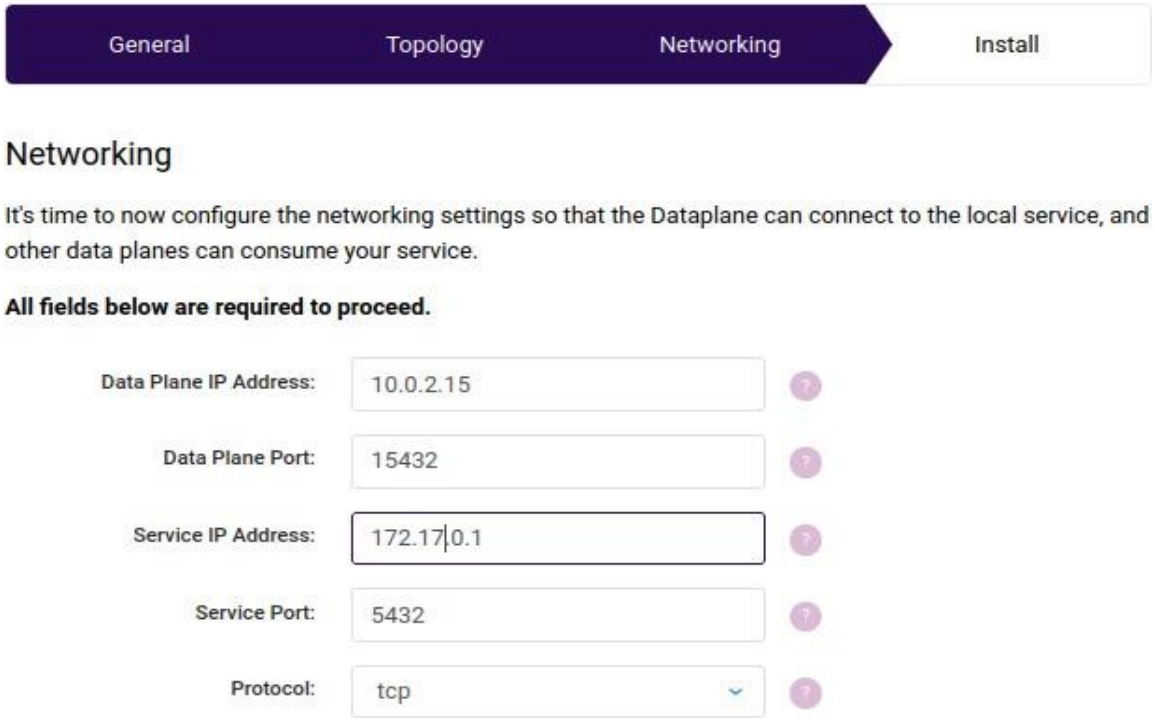

Figura 39: Configuração dos dados para a dataplane

Para finalizar, o Kuma faz o gerenciamento automático de *token*, que posteriormente será utilizado para que seja permitido a autenticação entre o plano de dados e o plano de controle, como é mostrado na fig. 40.

```
zenaida@zenaida-VirtualBox:~/kuma-1.3.1$ ./bin/kumactl generate dataplane-token --name=postegresql-jmzj5
l > kuma-token-postegresql-jmzj5l
zenaida@zenaida-VirtualBox:~/kuma-1.3.1$
```

Figura 40: Execução do token gerado para o PostgreSQL

Após a criação do token para a java app, o próximo a fazer é fazer o Kuma-run como é mostrado na fig. 41.

```
zenaida@zenaida-VirtualBox:~/kuma-1.3.1$ ./bin/kuma-dp run \
>        --cp-address=https://127.0.0.1:5678/ \
>        --dataplane="type: Dataplane
> mesh: default
> name: postegresql-jmzj5l
> networking:
>   address: 10.0.2.15
>   inbound:
>     - port: 15432
>       servicePort: 5432
>       serviceAddress: 172.17.0.1
>       tags:
>         kuma.io/service: postegresql
>         kuma.io/protocol: tcp" \
>        --dataplane-token-file=kuma-token-postegresql-jmzj5l
2022-05-28T16:09:51.610+0100    INFO    Skipping reading config from file
```

Figura 41: Executação do comando para fazer deploy do dataplane PostgreSQL

Como pode ser visto, na fig. 42 podemos ver todos os serviços ativos

Meshes › default › Data Plane Proxies ›

+ Create data plane proxy | Refresh

	Status	Name	Mesh	Type	Tags	Last Connected	Last Updated	Total Updates	Kuma DP version	Envoy version
✓	Online	java-ohrji8	default	Standard	KUMA.IO/PROTOCOL http KUMA.IO/SERVICE java	on 28/05/2022	on 12/07/2022	37378	1.3.1	1.18.4
View	Online	java1-hef19u	default	Standard	KUMA.IO/PROTOCOL http KUMA.IO/SERVICE java1	15 seconds ago	7 seconds ago	5	1.3.1	1.18.4
View	Online	postegresql-jmzj5l	default	Standard	KUMA.IO/PROTOCOL tcp KUMA.IO/SERVICE postegresql	on 28/05/2022	6 seconds ago	20820	1.3.1	1.18.4

Figura 42: Todos os serviços online

E assim conclui conforme descrito acima, a instalação e configuração dos microserviços do java e do PostgreSQL no Kuma. Realçando que a instalação e configuração não inclui nenhum recurso de segurança configurado.

6.5 Sumário

Neste capítulo, foi implementado e configurado um conjunto de serviços nomeadamente o PostgreSQL e o Java na framework do Kuma que contribui para a fase de análise e testes.
Essa experimentação consistiu em dois *front-end* (java/Java1) e o *backend* (PotgreSQL), com objetivo de avaliar o comportamento dos serviços.
No próximo capítulo, é abordado um conjunto de testes que permite a análise e avaliação da performance desses serviços.

Capítulo 7

Avaliação da performance dos serviços

Este capítulo documenta a configuração, avaliação e análise de seguranças destintas na performance dos serviços na framework do Kuma, de modo a avaliar a performance do Kuma.

7.1 Cenário

Os testes a realizar visam tornar um serviço com conjunto de microserviços mais seguro. Este serviço não suporta HTTPS no *frontend* e todas as comunicações com o *backend* também são feitas sem recurso a encriptação, conforme ilustrado na Fig.43.

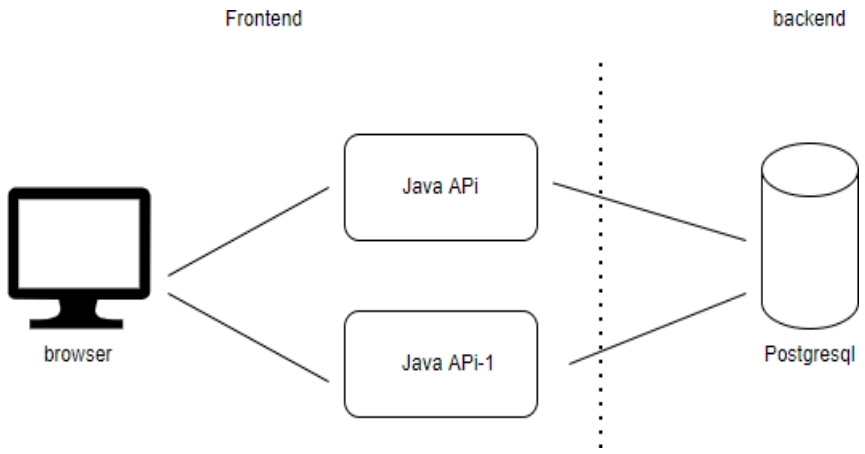

Figura 43: Planeamento e configuração de testes

Pretendia-se com a fig. 43 representar o cenário de configurações e testes. Onde o objetivo era abordar a nível de segurança e resiliência, como os serviços se comportam no Kuma, através do estudo e da análise da documentação das recomendações da NIST.

Neste cenário teríamos a comunicação entre o browser (utilizador) e as duas *front-ends* (o java API e java API-1). E como *backend* o postgreSQL.

Não foi possível concluir o plano de teste, afim de mostrar os resultados de uma forma pertinente, ficam assim, descritas, as configurações e implementações de uma forma teórica.

Junto com as políticas suportadas pelo Kuma, e pelas recomendações recomendadas pela NIST, documentadas no capítulo 3, pretendia-se através de um conjunto de configurações tornar o sistema o mais seguro, implementando as seguintes configurações:

1. Configuração do TLS mútuo entre o *front-end* (Java) e o *backend* (PostgreSQL)

2. Configuração de permissões de tráfego:
   Para o *backend* (PostgreSQL) configurar de modo que sejam apenas aceites tráfegos vindos do *front-end* (Java API, ou java API-1).
   Para o *front-end* (Java) configurar de modo que seja aceite comunicação de um determinado cliente.

3. Configuração do traffic route: permitir partilha de carga entre os *front-ends*. (Java API e Java API-1).

4. Configuração do *health check*, *circuit breaker* e *fault injection* na comunicação entre os *front-ends* e *backend*.

7.2 Configurações de segurança distintas na framework

Nesta parte pretendia-se implementar as configurações de segurança, descritas no cenário acima, ilustrado na fig.43.

1. Configuração do mTLS mútuo entre o *fron-tend* (Java API/Java API-1) e o *backend* (PostgreSQL)

A nível de configuração do mTLS, por padrão está desabilitado.
O Kuma apresenta um conjunto de políticas de segurança de recursos e configurações que ajudam a melhorar a segurança.
Por padrão a rede de comunicação entre os serviços não é segura nem criptografada, para isso o Kuma suporta habilitação da política do mutual TLS para fornecer autoridade de certificação dinâmico no recurso *default mesh* que vai atribuir de forma automática certificados de TLS aos serviços, especificamente aos proxies de planos de dados, que estão em execução ao lado dos serviços.

Como já foi visto, por padrão o mTLS não está habilitado, ou seja, o Kuma tem permissões de tráfego que permite que todos os tráfegos de todas as origens para todos os destinos se comunicam entre si.

A política do mTLS permite que todo o tráfego seja criptografado para todos os serviços da *mesh*, além de atribuir uma identidade a cada proxy de plano de dados.
Como visto anteriormente, descrito no capítulo 4, o Kuma suporta diferentes tipos de *back-ends* de CA (*certificate authority*), assim como rotação automática de certificados.

Para configurar o mTLS, o Kuma tem a opção de escolher um certificado integrado, ou optar por fornecer o nosso próprio certificado. Nessa configuração vai ser utilizado a configuração do mTLS com um *back-end* integrado, ou seja, o modo *Builtin*, criando automaticamente um CA para os serviços para toda a mesh.

2. Configuração da permissão de tráfego

A nível de segurança para a permissão de trafico, para o *backend* (PostgreSQL) configurar de modo que sejam apenas aceites tráfegos vindos do *frontend* (Java API, ou java API-1).
Para o *frontend* (Java) configurar de modo que seja aceite comunicação de um determinado cliente.
A permissão de tráfego permite determinar como os serviços java API e Java API-1 comunicam entre si.

O objetivo dessa configuração a nível de segurança, é permitir criar regras de permissão de acesso de modo a evitar acesso não autorizado.

3. Configuração do traffic route

A nível de segurança, com o traffic route, o objetivo é configurar uma política, ou seja, configurar regras de encaminhamento.
Ou seja, um conjunto de configurações que indica um conjunto de regras que permite que a carga de informação seja partilhada entre os *front-ends* javaAPI e o javaAPI-1.

O objetivo dessa configuração é permitir encaminhar a comunicação da javaAPI para Java API-1 de modo que a informação seja partilhada entre eles.

4. Configuração da política de segurança como o *health check, circuit breaker*

O objetivo de configurar o *circuit breaker,* seria implementar a estratégia de falhas de modo a avaliar a resiliência do Kuma na perspetiva de causar falhas e analisar como o Kuma reagiria perante essas falhas, e analisar também de modo a não permitir a entrega de dados ao

microserviço que está falhando. Ou seja, a estratégia seria corromper um dos serviços, e o cliente fazer solicitações.

Aproveitando desse cenário, outra implementação seria o *healthCkech*, com o objetivo de verificar a integridade dos serviços e modo a minimizar o número de solicitações no caso do serviço que estaria corrompido, ou temporariamente inacessível.

Tento em conta o conhecimento e análise da existência de outras políticas de segurança, para este plano de teste e análise da segurança e resistência, foram escolhidas as políticas conforme descrita acima.

Capítulo 8

Conclusão

Neste capítulo serão apresentadas as conclusões e os desafios encontrados ao longo do desenvolvimento desta dissertação.

A grande escala do alcance da internet, assim como a velocidade de como a tecnologia, e as aplicações nativas da nuvem - *cloud*, têm proporcionando tanto para as empresas, quanto aos utilizadores novos patamares de qualidade.

Neste meio tecnológico em que grandes empresas têm seus serviços online, estão aptos a qualquer tipo de interveniência, a nível de segurança principalmente. A *service mesh*, vem mostrando e provando ser uma tecnologia promissora para a próximas gerações e capaz de solucionar esses problemas, afim de fornecer uma comunicação segura entre as aplicações e serviços. Empresas como a Netflix, Amazon, vêm já adotando esta abordagem de modo a suprir as suas necessidades, em termos de segurança, alta escalabilidade, e disponibilidade.

Ao longo do desenvolvimento deste trabalho, foi feito uma série de análises de forma sistemática da abordagem da *service mesh*, uma tecnologia emergente, que visa reduzir a complexidade operacional entre os aplicativos e a comunicação entre os microserviços.
Assim sendo, a primeira contribuição para este trabalho foi uma análise sistemática de um conjunto de políticas e recomendações em nível de segurança, que permite uma avaliação objetiva do suporte de segurança.

Pelo fato de não existir um padrão único da implementação de uma *service mesh* e sim várias opções abrangendo soluções académicas e comerciais, são inúmeras as propostas encontradas na tentativa de solucionar as várias necessidades. Este trabalho contem informações relevantes que ajudam na escolha da framework para implementar a *service mesh*, sobretudo numa vertente de segurança.

Ao longo de desenvolvimento dessa dissertação houve bastantes desafios, principalmente na parte prática, pois houve uma certa escassez de documentação que abrange o componente pratico.
Não foi possível concluir todos os objetivos, principalmente a parte da análise do suporte de resiliência de microserviços, porém com a análise e estudo teórico, ficou descrito quais seriam as estratégias de implementações capazes para uma boa avaliação.
Porem, pode se dizer, que a *service mesh*, é um paradigma muito importante, bastante promissora, que com certeza pelas funcionalidades, que ele dispõem em ser capaz, vai ser uma opção bastante requisitada pelas grandes empresas que procuram manter seus serviços seguros. Sendo a segurança, um requisito crucial para o desenvolvimento e comunicação dos serviços.

Referências

[1] https://kubernetes.io/

[2] https://www.xtivia.com/blog/challenges-of-microservices-and-service-mesh/

[3] P. Jamshidi, C. Pahl, N. C. Mendonc¸a, J. Lewis, and S. Tilkov, “Microservices: The Journey So Far and Challenges Ahead,” IEEE Software, vol. 35, no. 3, pp. 24–35, 2018.

[4] Khan, “Key Characteristics of a Container Orchestration Platform to Enable a Modern Application,” IEEE Cloud Computing, no. 5, pp. 42–48, 2017

[5] https://queue.acm.org/detail.cfm?id=3277541

[6]https://landscape.cncf.io/card-mode?category=service-mesh

[7]https://platform9.com/blog/kubernetes-service-mesh-a-comparison-of-istio-linkerd-and-consul/

[8]https://thenewstack.io/survey-results-service-mesh-useful-for-security-observability-and-traffic-control/

[9]https://www.zappts.com/blog/arquitetura-monolitica-e-microsservicos/

[10]https://www.g2.com/products/hashicorp-consul/reviews

[11]https://doc.traefik.io/traefik-mesh/

[12]https://doc.traefik.io/traefik-enterprise/operations/service-mesh/

[13]https://blog.lsantos.dev/uma-introducao-a-service-mesh-com-linkerd/

[14] https://www.cloudflare.com/learning/access-management/what-is-mutual-tls/

[15] https://github.com/traefik/mesh

[16] https://www.getambassador.io/learn/service-mesh/introduction-to-service-mesh-interface/

[17] https://www.hashicorp.com/products/consul

[18] https://aws.amazon.com/pt/app-mesh/?aws-app-mesh-blogs.sort-by=item.additionalFields.createdDate&aws-app-mesh-blogs.sort-order=desc&whats-new-cards.sort-by=item.additionalFields.postDateTime&whats-new-cards.sort-order=desc

[19] https://aws.amazon.com/pt/app-mesh/faqs/

[20] https://konghq.com/blog/getting-started-Kuma-service-mesh/

[21] https://grpc.io/

[22] https://www.toptal.com/kubernetes/service-mesh-comparison

[23] https://servicemesh.es/

[24] https://dzone.com/articles/the-service-mesh-in-the-microservices-world

[25] https://www.weave.works/blog/what-are-microservices/

[26] https://docs.aws.amazon.com/index.html?nc2=h_ql_doc_do_v

[27] https://servicemesh.es/

[28] X. Zhou, X. Peng, T. Xie, J. Sun, W. Li, C. Ji, and D. Ding, "Delta Debugging Microservice Systems," in Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18). ACM, 2018, pp. 802–807.

[29] https://www.zappts.com/blog/arquitetura-monolitica-e-microsservicos/

[30] https://Kuma.io/.

[31] https://www.vmware.com/topics/glossary/content/kubernetes

[32] https://www.redhat.com/en/topics/microservices/what-is-a-service-mesh

[33] https://thenewstack.io/survey-results-service-mesh-useful-for-security-observability-and-traffic-control/

[34] https://www.toptal.com/kubernetes/service-mesh-comparison

[35] https://smi-spec.io/

[36] https://prometheus.io/docs/introduction/overview/

[37] https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=8705911&casa_token=hvCoxz0fQD4AAAAA:8BDufqkOgJXIaM0QC5-GncJe-5d40xGdC1i0CItrbvW4npP-2OsYuUU6NUBm7vkP2VqYHdmVELw&tag=1

[38] https://www.infoq.com/br/articles/service-mesh-ultimate-guide/

[39] https://www.linkedin.com/pulse/tls-mtls-demystified-muhammad-atif-rafiq

[40] https://thenewstack.io/mutual-tls-microservices-encryption-for-service-mesh/

[41] https://www.f5.com/labs/articles/education/what-is-mtls

[42] https://docs.aws.amazon.com/IAM/latest/UserGuide/introduction.html

[43] https://www.aplyca.com/es/blog/service-mesh-arquitectura-de-microservicios

[44] https://istio.io/latest/about/service-mesh/

[45] https://linkerd.io/2017/04/25/whats-a-service-mesh-and-why-do-i-need-one/

[46] https://www.opus-software.com.br/service-mesh-e-a-comunicacao-dos-microsservicos/

[47] https://www.envoyproxy.io/

[48] Normas propostas pelo NIST Special publication 800-204A, Building Secure Miroservices-based Applications Using Service-mesh Architecture. Available from: https://doi.org/10.6028/NIST.SP.800-204A

[49] https://Kuma.io/docs/1.3.1/installation/ubuntu/#_1-download-Kuma

[50] https://Kuma.io/docs/1.3.x/quickstart/kubernetes/#prerequisites

[51] https://www.toptal.com/nodejs/why-the-hell-would-i-use-node-js

[52] https://www.haproxy.com/

[53] https://docs.microsoft.com/en-us/cpp/cpp/namespaces-cpp?view=msvc-170

[54] https://spiffe.io/

[55] https://www.controle.net/faq/san-storage-area-network

[56] "Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center", Anais do 8º Simpósio USENIX em Projeto e Implementação de Sistemas em Rede ser. INDE '11 , vol. 11, disponível em: https://www.usenix.org/legacy/event/nsdi11/tech/full_papers/Hindman.pdf

[57] https://datatracker.ietf.org/doc/html/rfc5280#section-4.2.1.9

[58] https://acervolima.com/a-historia-da-netflix-e-dos-microsservicos/

Apêndices